MASTER IN COMPUTER SCIENCE

Program analysis for concurrent programming
consistency verification of STM Haskell programs using Moth

Duc, Aurélie

Award date:
2014

Awarding institution:
University of Namur

Université de Namur
Faculté d'informatique
Année académique 2013–2014

Program analysis for concurrent programming: consistency verification of STM Haskell programs using Moth

Aurélie Duc

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Wim Vanhoof

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

ACKNOWLEDGMENTS

I would like to thank Prof. Vanhoof for his assistance and corrections throughout the writing of this paper.

My sincere thanks also goes to FX for his tips for the grammatically challenged, analyse of my relationship with commas and general after-sale service, to my baby sister for proof-reading and pretending to enjoy some chapters, to my friends and teammates for their support and to my boyfriend and family (including you, Moody) for unwavering support, emergency proof-reading and kicks.

Contents

Introduction

Software verification is a part of computer science whose goal is to ensure that a piece of software is correct (with respect to its specification), by testing or formally proving the program correct.

There are two aspects in *correctness*: partial correctness and termination. An algorithm is *partially correct* if it has the property of giving a correct answer (with respect to its specification) if giving any. An algorithm is *totally correct* if it is partially correct and can be shown to always terminate.

Both aspect of correctness are just as important to produce software one can really rely on, and correctness is just as essential on every type of system. However, partial correctness verification for software running on concurrent systems is a specific field of study, and one which has been the object of feverish activity in the last fifteen years. The generalisation of multicore and multiprocessor computers has led to a crucial need to ascertain the reliability of programs developed to gain individual efficiency benefits from concurrency.

Program verification for concurrent programming obviously encompasses all the difficulties of sequential program verification, but it also introduces new difficulties. Not the least of those difficulties is the management of variables shared by several threads in a program.

In order for a program to benefit from concurrency on an individual basis, it needs to spawn several threads to perfom separate tasks. However, since those threads will execute in the same global memory state, it is necessary to make sure that the global memory state remains consistent regardless of the interleaving of operations from each thread.

In the context of this work, *concurrent programming* must be understood as a programming paradigm in which different computations originating from a single run of a program are potentially executed at the same time (*concurrently*) instead of one after the other (*serially*). The possible interactions discussed are those that may occur solely regarding the program variables.

Consistency can be understood as a property expressing the fact that the values of all the variables are coherent. At the end of one logical atomic operation (regardless of whether it is implemented as one or several successive instructions) involving changes to several variables, all or none of them must be updated for consistency to be maintained by the operation. In addition, the program should not be unintentionally non determinist, which implies that its results should not depend on thread interleaving.

Consistency violations will mostly affect partial correctness, making them twice as hard to detect as other kinds of errors. Indeed, (partial) incorrectness is inherently more vicious than non-termination because, while one would probably notice if a program does not terminate, an erronerous answer may very well go unnoticed. On top of that, an error resulting from thread interleaving will most likely be caused by specific thread

interleavings that may occur very rarely in practice, making it extremely difficult to detect through "regular" testing.

Because of the difficulties mentioned above and of how hard it is to reason correctly about concurrent threads when analysing a piece of code, it is natural for programmers to wish for concurrent software verification to be automated.

An ideal automatic checking tool would warn a programmer about every error in his code (a quality that will be referred to as *completeness*) but only real errors (a quality that will be referred to as *soundness*). Moreover, the verification tool should not be too "expensive" to use, which means, according to [1], that it should neither affect the run time of the program, nor take too long to perform the analysis at compile time, and also that it should not require the user to spend a lot of time providing program annotations.

The inexpensiveness requirement is in direct competition with the completeness and soundness requirements. Indeed, by ruling out the option of asking the programmer to document his design choices through annotations, it forces the checker to make assumptions on what an expected behavior is instead of being able to get that information from a reliable source. Nevertheless, the inexpensiveness is essential to achieve widespread adoption of the checker.

Accordingly, inexpensive checkers are forced to test code for violation of general "good practice" criteria and to report every violation found without having the possibility of finding out whether the code programmer really intended to do what he did or if he made a mistake. They are therefore very likely to report a violation in some cases where there is no real error. Conversely, they should not report violations of properties that will very rarely be symptomatic of errors because an abundance of spurious warnings will also discourage some users. They are hence very likely not to report a violation when the programmer has made a mistake in an unusual way.

The need for testable and formally defined property to express scenarios that indicate consistency errors "most of the time" therefore leads the checkers to be often both too demanding (which results in *false positives*) and not demanding enough (which results in *false negatives*). In that context, a *false positive* occurs when a checking tool reports a piece of code as erroneous when it cannot cause consistency violations, and a *false negative* occurs when a checking tool does not report a piece of code as erroneous despite the fact that it could cause consistency violations.

Transactional memory is an elegant and composable syncronisation mechanism to guarantee correctness in a concurrent environment. It takes the responsibility of providing atomicity and isolation where the application programmer declares that he expects it by identifying a group of operations as a *transaction*. However, it does not exempt the application programmer from the task of synchronizing adequately. The transactional memory mechanism may provide atomicity and isolation to transactions but it cannot guarantee memory consistency as that characteristic depends on the correct definition

of transactions by the programmer.

Haskell is a functional language well-suited for concurrent programming because side effects are inherently limited through its type system. Its transactional memory extension, STM Haskell, is the basic model of concurrent programming environment for the remaining of this work.

The *Moth tool*, presented in [2], is an automatic checker for different types of consistency errors in transactional memory Java programs. It is not the ideal tool dreamed of by programmers but is very useful in eliminating some kinds of concurrency errors at limited cost for the user, because its authors have chosen to give priority to inexpensiveness at the expense of soundness and completeness. This paper discusses the adaptation of Moth to STM Haskell programs.

The first part of this work presents its context. The first chapter outlines what concurrent programing in STM Haskell is by presenting a few fundamental notions on Haskell. The second chapter defines what transactional memory is, how it works and how it is implemented in Haskell. The third chapter presents different types of concurrency related issues, defines them and analyses their possibility of occurence in STM Haskell.

The second part presents the adaptation of Moth to STM Haskell. Moth uses a two-step evaluation procedure: generating an abstract model of the concurrent operations taking place in the program, and then analysing the generated model for indication of possible concurrency errors. The fourth and fifth chapters therefore present the adaptation of the two steps of the analysis to Haskell. The sixth chapter then illustrates the entire Moth analysis works on a STM Haskell program example.

The last part briefly analyses other consistency violations checkers to compare their methodology and the range of errors they would detect in STM Haskell programs with Moth. And finally, it also discusses how some ideas from other tools could be used to improve the efficency of the Moth tool for STM Haskell.

Part I

Concurrent programming in STM Haskell

Chapter 1

Haskell: a few fundamental notions

The present chapter is mainly adapted from [3] and [4]. It provides a few basic reminders on the Haskell programming language to help the reader to understand later examples. It also expands a bit on how its purity makes it suitable for concurrent programming and how that purity is enforced through the type system.

1.1 Haskell: a functional programming language

Functional languages such as Haskell are well-suited for concurrent programming because the parts of a program that may generate side effects (and potentially concurrency errors) are well identified and generally limited.

1.1.1 Functional language and purity

Keeping things simple, a *functional language* is a language in which a program is composed of expression evaluations, whereas an imperative language program is mostly composed of actions (sequences of commands).

An *expression* represents a value (a typed entity often represented by an identifier). It is either a variable (a name representing an expression), a basic value, a function call or a combination of variables, values and operators. An expression can be *reduced* or *evaluated* to obtain the value it represents.

A function evaluation returns a value but does not modify its arguments, so pure functions cannot "interfere" with one another or with their environment. Thus, purely functional programs can be relied on not to have side effects.

A function has *side effects* if it has interactions with the "world" such as reading from or modifying a global variable, or reading from or writing to a file (or even to any input or output channel). Such a function is called *impure* while a *pure* function is a function that never has side effects and therefore never alters the state of the program (or the "world").

Purely functional (sub)programs are easier to reason about and compose into larger programs because no context needs be considered when reasoning about their correctness. Consequently, functional (sub)programs are well-suited for concurrent programming because they don't tamper with one another. On the contrary, impure (sub)programs can behave differently in different environments and may alter the global state of the program so they shouldn't be used without considering interferences as well from than to other (sub)programs.

1.1.2 Haskell type system

Despite being primarily a functional language, Haskell allows its users to use impure functions but provides a way to identify and separate pure functions from impure ones through the type system, so that the user is not personally responsible for keeping track of which functions can cause problems because of their side effects. A specific `IO a` type exists for impure functions, and Haskell guarantees that functions that are not typed as such are all pures.

An expression in Haskell always has a *type* that can either be built-in or user-defined (by renaming, enumeration, union or products of types). The most common built-in types are the basic types such as Int, Integer, Float, Double, Bool and Char and the standard compound types: list and tuples. A *tuple* is a fixed-size collection of values where each value can have a different type and is denoted $(\texttt{el}_1, \texttt{el}_2, \ldots)$, and a *list* is a "set" of elements of the same type and is denoted $[\texttt{el}_1, \texttt{el}_2, \ldots]$.

A peculiarity of Haskell is that a type can be *polymorphic* if it contains *type parameters*. For example, `Tree a` is a tree whose nodes have the type `a` where `a` is the type parameter and can be any type. The `IO a` type used for impure function is such a type because the actual return type of an impure function can be anything, including nothing if the function does not return anything, which will be denoted $()$. We can think of an impure function with type `IO a` as a function performing some impure operations before evaluating to some value of type `a`.

1.1.3 Functions

In the previous subsection, we stated that a functional program is composed of function evaluations. Some functions are built-in, but most functions used in a program are user-defined. So Haskell programs are mostly composed of function definitions and function evaluations.

Function definition

A *function* is a relation that associates exactly one output to each value of its domain.

The function $\mathtt{f}$ with parameters $\mathtt{x_1}, ..., \mathtt{x_p}$, $\mathtt{p} \geq 0$ is defined by

$$
\begin{array}{lll}
\mathtt{f} :: \mathtt{type_1} \quad \ldots \quad \rightarrow \mathtt{type_p} \rightarrow \mathtt{type_{p+1}} & & \\
\mathtt{f}\ \mathtt{x_1} \ldots \mathtt{x_p} & & \\
\quad \mid \mathtt{C_1} & = & \mathtt{e_1} \\
 & & \ldots \\
\quad \mid \mathtt{C_k} & = & \mathtt{e_k}
\end{array}
$$

where $\mathtt{k} \geq 1$ and $\forall i \in \{1, \ldots, p\} : \mathtt{x_i}$ has type $\mathtt{type_i}$ and $\forall j \in \{1, \ldots, k\} : \mathtt{C_j}$ is a condition and $\mathtt{e_j}$ has type $\mathtt{type_{p+1}}$. The first part of the definition, $\mathtt{f} :: \mathtt{type_1} \ldots \rightarrow \mathtt{type_p} \rightarrow \mathtt{type_{p+1}}$ is called the *function signature*.

The function f is called using $\mathtt{f}\,\overrightarrow{x}$ where $\overrightarrow{x} = \mathtt{x_1}\,\mathtt{x_2}\,\ldots\,\mathtt{x_p}$ are actual parameters.

It should be noted that the types of the parameters of a function as well as its output type can be anything, including compound types, user-defined types, functions or *type variables*.

Higher-order functions

A function of functions (or *higher-order function*) is a function that has a function as at least one of its parameters.

An example of such a function in Haskell is `twice`, returning the composition of a function with itself.

$$
\begin{array}{lll}
\mathtt{twice} :: (\mathtt{Integer} \rightarrow \mathtt{Integer}) & \rightarrow & \mathtt{Integer} \rightarrow \mathtt{Integer} \\
\mathtt{twice\,f\,x} & = & \mathtt{f\,fx}
\end{array}
$$

Calling `twice` using the function `square` and the Integer 2 as actual parameter would return $\mathtt{twice\ square}\ 2 = \mathtt{square\ square}\ 2 = \mathtt{square}\ 2 * 2 = 2 * 2 * 2 * 2 = 16$ if

$$
\begin{array}{lll}
\mathtt{square} :: \mathtt{Integer} & \rightarrow & \mathtt{Integer} \\
\mathtt{square\,x} & = & \mathtt{x} * \mathtt{x}
\end{array}
$$

Polymorphic functions

A function that has a *type variables* such as `a` as at least one of its parameters is called *polymorphic*. Such a function doesn't care what actual type it is called with and operates in the same way no matter what type its arguments are.

An example of of polymorphic function in Haskell is `fst` returning the first element of a pair.

$$\begin{aligned} \texttt{fst} :: (\texttt{a}, \texttt{b}) &\rightarrow \texttt{a} \\ \texttt{fst}\,(\texttt{x}, \texttt{y}) &= \texttt{x} \end{aligned}$$

Calling `fst` using $(1, 2)$ as actual parameter would return `1`, calling it using $(\texttt{True}, \texttt{False})$ as actual parameter would return `True` and calling it using $(1, \texttt{False})$ as actual parameter would return `1`.

Operators

An *operator* is a binary function written between its arguments.

Haskell lets users define their own operators but also provides the most common built-in operators. For example, a comparison is made with $==$(equality), $/=$ (inequality) or with one of the comparison operators $>=$, $<=$, $<$, $>$. There are also the logical `or`, `and`, $|||$ and && operators and the basic arithmetic operators $+, -, /$ and $*$.

1.1.4 Control structures

Since Haskell is composed of expression evaluations and not of instruction executions, there is no need for control structure to control program flow. However, there are a few choice structures that can be used to define the value of an expression using multiple alternatives.

Choice As well as the condition guarded definitions seen in the function definition, there is an $\texttt{if}\,\texttt{e}_1\,\texttt{then}\,\texttt{e}_2\,\texttt{else}\,\texttt{e}_3$ construct to choose between two expressions where $\texttt{e}_1$is a boolean and $\texttt{e}_2$ and $\texttt{e}_3$ are expressions that can have any type.

Case Another way of choosing between different values for an expression is to use the case construct. The result of that construct is the expression associated with the first pattern to match starting from top case:

```
case Name of
   pattern_1  →  e_1
              ⋮
   pattern_n  →  e_n
```

Loops There are no built-in loop instructions in Haskell such as the *for* or *while* loops in most imperative languages. Repeated computations are implemented using recursive functions and often moving through lists. However, Haskell supports user-defined control structures so it is possible to define a loop function just as any other function.

1.2 Haskell: an impure langage

In the previous section the purely functional, side-effect free part of Haskell, was briefly presented. However, it is often necessary for a program to perform some "inpure" work such as reading input data or writing output. It is of course possible to do I/O operations in Haskell but it is done in a secure way : I/O actions are typed in a particular manner so that impure code can only be found in functions whose signature indicates clearly they are inpure, which ensures that side effects can only be caused by those functions, while the pure ones are completly safe. The current section is adapted from chapter seven of [4].

1.2.1 Actions

A function with return type `IO a` is an I/O *action*, an impure function. The `a` in the type definition is the output type of the function if it returns a value, or () if it does not.

I/O actions only produce an effect when performed inside another I/O action, so the main function of a program needs to have the type `IO ()` (the empty tuple () indicating there is no return value) if the program is to have any kind of interactions with the outside world. A interesting consequence of this requirement for nested I/O types is that if a function is not I/O-typed, there is not need to check its code for enclosed impure functions: freedom from side-effects is guaranteed.

Classic read and write

A very common example of I/O actions are Haskell's functions to read from the standard input and write to the standard output. They all have an `IO` type because the type system only allows side effects in I/O actions.

`putChar` :: `Char` $->$ `IO`() takes a Char and returns an I/O action that prints the Char on the standard output, `putStrLn` :: `String` $->$ `IO`() takes a string and returns

an I/O action that prints the string on the standard output, `getChar` :: `IO Char` is an I/O action that reads a character from the standard input and delivers it as a Char and `getLine` :: `IO String` does the same with a string.

Mutable references

Another common impure operation is to write to or read from a global variable. Haskell standard type for such mutable variables is `IORef a` and operations on `IORef a` also need to have an *IO* type.

For example, a new global variable of type `a` is created using `newIORef` :: $\texttt{a} -> \texttt{IO}\,(\texttt{IORef a})$, `readIORef` :: $\texttt{IORef a} -> \texttt{IO a}$ is used to read from an `IORef` and `writeIORef` :: $\texttt{IORef a} -> \texttt{a} -> \texttt{IO}\,()$ to write to it.

1.2.2 Specific I/O constructs

While we said earlier that there is no need for flow control in pure functional Haskell, it does no hold true for I/O actions, so there are a few language constructs that are specific to actions.

For example, I/O actions can be "glued" together to define a series of consecutive actions to perform using the *bind combinator* `do`: $\texttt{do}\{\texttt{S};\ldots;\texttt{S}\}$.

Another need arising from I/O action usage is to enable pure and impure functions to communicate with one another and to be composed. The $<-$ operator is used to store the result of an I/O operation in a variable ($\texttt{IO a} \rightarrow \texttt{a}$) and conversely the `return` operator is used to return the value of a pure computation as an I/O type ($\texttt{a} \rightarrow \texttt{IO a}$).

1.3 Concurrent operations in Haskell

As discussed in the first section, Haskell is well-suited to concurent programming since most of the Haskell code will be pure, and therefore the operations executing concurently won't interfere with one another.

A *thread* in Haskell is an I/O action that executes independently from other threads. To create a thread, we use $\texttt{forkIO} :: \texttt{IO a} -> \texttt{IO Thread}_{\texttt{Id}}$. Typically, threads communicate through global variables typed `MVar` but we will see, in the following chapter, that other types of global variables may also be used.

Chapter 2

Software transactional memory in Haskell

The previous chapter presented a commonly used type for variables shared between several threads, `MVar`. Mutual exclusion on those variables is traditionally enforced using locks. However, locks-protected `MVar a` is not the only secure way to share variables or, for that matter, the easiest one, and we will use a Haskell polymorphic type called `TVar a` instead. We will use STM Haskell as our basic model of concurrent programming environment for the remaining of this paper.

The first section of this chapter briefly discusses the issues raised by the lock-based approach to explain why that traditional approach was not chosen. The second section presents the idea of software transactional memory behind the `STM a` type, its avantages and how it can be implemented. The last section presents the type in itself and its usage in concurrent programs.

2.1 Lock-based synchronisation

The basic pitfall concurrent programs need to dodge is the possibility for two threads to access a variable at the same time in such a way that the result is dependent on thread shelduling: such a problem is called a low-level data race and is defined as follows in [5]: "A *low-level data race* can occur when two concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous" [1].

A classic example of low-level data race is that of two threads both attempting to increase the value of a variable x by one: if the first thread reads the initial value x_0 then

[1]Cyrille Artho, Klaus Havelund and Armin Biere, "High-Level Data Races" in *Software Testing, Verification and Reliability: Special Issue: VVEIS 2003 Workshop on Verification and Validation of Enterprise Information Systems, 13(4)*, December 2008, page 1, retrieved October 2012 from http://staff.aist.go.jp/c.artho/papers/stvr03.pdf [5].

increases x to $x_0 + 1$ before the second thread does its reading, the resulting value of x will be $x_0 + 2$ whereas if the second thread reads the value of x before the first thread has updated it, both threads will read x_0 and the resulting value of x will be $x_0 + 1$.

Low-level data races are traditionally avoided through usage of synchronisation methods such as locks. A *lock* is a shared variable used to enforce mutual thread exclusion on access to a resource: a thread can only access the ressource guarded by a lock if no other thread has previously "locked" (and not released) it, the state of locking being materialised by the value of the lock. The part of the code defined by a lock acquisition and and a lock release is called a *critical section*.

Usage of critical sections affects a concurrent system's performance: one thread halted or delayed in the critical section due to page fault, preemption,... may cause other, non faulty threads to be unable to progress. Therefore, in order to write programs that truly benefit from concurrency in terms of performance, it is suitable to limit the size and number of critical sections as much as possible.

Indeed, big critical sections limit the risk of concurrency related errors but they do so by effectively limiting concurrency in itself. Conversely, multiple small critical sections are more efficient because one thread won't limit the action of other threads for very long while executing but may induce subtle concurrency bugs such as those we will discuss in chapter 3.

Nevertheless, there remains the need for shared variables to be dealt with in such a way that memory is never inconsistent. As a consequence, concurrent programming using critical sections gives rise to complicated design choices to guarantee both performance and correctness and makes subprogram composability tricky because the wrong order in lock aquisition can cause deadlocks. A *deadlock* occurs in a concurrent system when all threads are blocked, each waiting for some action by one of the other threads or a ressource held by one of the other threads so none of them can progress and the program can't terminate.

2.2 Software transactional memory

The objective of *transactional memory* is to hide synchronisation mechanisms from the application programmer by providing him with a composable langage construct to simply define a transaction and leave the responsibility of the transaction's correct execution to a a lower-level mechanism.

Accordingly, software transactional memory implementations would be expected to assume the burden of providing correctness guarantees with regard to concurrency so that it does not depend on each program's locking discipline.

There are both software and hardware primitives to define a transaction but we will only discuss the software version implemented in Haskell that will be used in the re-

maining of the present paper.

The first part of this section will elaborate on what software transactional memory is, the second will detail its avantages and the last part will present possible implementations.

2.2.1 Definitions of software transactional memory

Software transactional memory (STM) is a synchronisation mechanism defined in order to avoid critical section usage. It is defined using the notion of transaction originating in database theory. A programer defines transactions containing operations that his program needs to execute atomically and the STM implementation guarantees the transactions' atomicity.

A *transaction* is a finite sequence of instructions executed by a thread that has the property of being *atomic* and *isolated*: the sequence is either completly executed (the transaction *commits*) or not at all (the transaction *fails*). Therefore, they appear to the external world as if they took effect instantaneously and were all executed serially.

In [6], software transactional memory is defined as "a shared object which behaves like a memory that supports multiple changes to its addresses by means of transactions".[2]. In [7], it is more precisely defined as "a low-level application programming interface for synchronizing shared data without using locks. Transactional memory supports a computational model in which each thread announces the start of a transaction, executes a sequence of operations on shared objects, and then tries to commit the transaction. If the commit succeeds, the transaction's operations take effect; otherwise, they are discarded" [3].

2.2.2 Advantages of software transactional memory

When programming with transactional memory, the application programmer specifies a transaction grouping operations he wants to execute atomically, and the STM implementation has the responsibility to guarantee the transactions' atomicity.

STM hides the complexity of the synchronisation mechanism so the programmer does not need to care about multiple lock acquisitions, progress guarantee and such difficulties. By taking the responsibility of providing the atomicity the programmer declares, software transactional memory allows him to think about atomic operations at the level of the program semantics.

[2]Shavit, N and Touitou, D, *Software transactional memory*, in Proceedings of the fourteenth annual ACM symposium on Principles of Distributed Computing, Ottowa, ACM, 1995[6].

[3]Herlihy, M and Luchangco, V and Moir, M and Scherer, W, *Software transactional memory for dynamic-sized data structures*, in Proceedings of the 22nd annual ACM symposium on Principles of Distributed Computing, ACM, 2003[7].

The optimistic execution model used by most transactional memory implementations (and specifically by Haskell stm package) is also more efficient in general. Indeed, conflicts seldom occur in programs, and since STM provides mechanisms to deal with them when they do, it appears to provide both performance and correctness guarantees. The execution model is called *optimistic* because it does not unable concurrent access for fear they cause conflicts as locks do, but only aborts one of the concurrent transactions when it is found to have caused conflict.

Transactions are also easier to compose into bigger transactions without causing deadlocks that lock protected accesses because the programmer does not need to worry about lock acquisition order.

2.2.3 Implementations of software transactional memory

The current subsection provides a few basic notions on possible pratical implementations of software transactional memory ouside of Haskell. While we will use `STM a` as a black box in the remaining of the present text, those implementation examples provide an useful insight into the inner workings of software transactional memory.

In [6], the authors present an STM implementation whose idea is to perform memory changes through transactions which keep a copy of the data at the beginning of their execution ("old"), modify another copy ("new") and commit by changing the value in memory to "new" if they find it is still equal to "old" (using an atomic kword Compare&Swap) and fail and restart otherwise.

They show their implementation is non-blocking because they use a helping policy between threads to make sure that all transactions complete (complete, not commit!) even if its native process has been swapped out, has crashed or is delayed where "an STM implementation is is said to be *non-blocking* if the repeated execution of some transaction by a process implies that some process (not necessarily the same one and with a possibly different transaction) will terminate successfully after a finite number of attempts in the whole system"[4].

Their implementation, however, only supports static transactions (it requires the program to declare in advance the memory locations it will access).

The mechanism is that a transaction modifies a private copy of an object without further synchronisation and only makes its change visible to other transactions when it commits. If another transaction accesses the original object concurrently, the STM resolves the conflict by aborting one of the transactions. No copy is made for a write-only access but a transaction only commits if no concurrently executing transaction has modified an object it has read.

[4]Shavit, N and Touitou, D, *Software transactional memory*, in Proceedings of the fourteenth annual ACM symposium on Principles of Distributed Computing, Ottowa, ACM, 1995, page 6, retrieved November 2012 from www.cse.ohio-state.edu/ agrawal/788-su08/.../shavit95software.pdf [6].

In [7], an implementation that supports dynamic transactions is presented. It uses transactional objects with three fields: old, new and a field indicating the last transaction (with a status field indicating which of old or new is the "real" object). A *transactional object* is defined as "a container for a regular object. A transaction can access the contained object by opening the transactional object, and then reading or modifying the regular object. Changes to objects opened by a transaction are not seen outside the transaction until the transaction commits. If the transaction commits, then these changes take effect; otherwise, they are discarded." [5].

The implementation in [7] is only obstruction-free ("A synchronisation mechanism will be said to be *obstruction-free* if any thread that runs by itself for long enough makes progress, which implies that a thread makes progress if it runs for long enough without encountering a synchronization conflict from a concurrent thread" [6]).

The risk of a halted thread blocking others is therefore excluded, but it is possible for concurrent threads to repeatedly keep one another from committing. However, progress is managed by an abortion mechanism supervised by a plugged-in contention manager: a transaction can abort another but, before it does, it consults the contention manager to know if it should proceed or give the other transaction time to complete. Blockage is thus excluded if the manager policy ensures that a transaction that asks "sufficiently many times" permission to abort another is eventually granted it.

The implementation also integrates different opening modes (so that read-only transactions do not keep one another from commiting) as well as the idea of validating a transaction whenever it opens a transactional object (checking whether any object opened by the transaction has since been opened in a conflicting mode by another transaction to keep a transaction from observing inconsistent states between objects already opened and new objects it tries to open). That last feature allows a transaction to detect that it won't be able to commit in the future to avoid wasting time.

Another, potentially risky, particularity of the implementation is the ability to release objects from a transaction before it commits.

Even if that implementation supports a larger set of transactions than the one presented in [6], it does have troublesome limits: it does not support nested transactions while the posibility of composing transactions neatly is normally one of their advantages over locks. A nested transaction is defined in [8] as a transaction that begins and ends within the scope of a surrounding transaction.

[5]Maurice Herlihy, Victor Luchangco, Mark Moir and William N. Scherer III, "Software Transactional Memory for Dynamic-Sized Data Structures" in *Proceedings of the 22nd annual ACM symposium on Principles of Distributed Computing*, 2003, page 2, retrieved November 2012 from cs.brown.edu/courses/csci1610/papers/stm.pdf [7].

[6]Maurice Herlihy, Victor Luchangco, Mark Moir and William N. Scherer III, "Software Transactional Memory for Dynamic-Sized Data Structures" in *Proceedings of the 22nd annual ACM symposium on Principles of Distributed Computing*, 2003, page 2, retrieved November 2012 from cs.brown.edu/courses/csci1610/papers/stm.pdf [7].

In this case, the mechanism is that a transaction opens the transactional object to access the contained object, then modifies a private copy of an object without further synchronisation, and only makes the changes visible to other transactions when it commits. Only one transaction can open a transactional object in write mode at a time, but problematic concurrent access can occur through reads, so each time a transaction opens an object in read mode, it checks that no object it has read before has since been modified to avoid reading an inconsistant global state ("validating the transaction"). Therefore, commiting a transaction is only possible if no conflicting transaction has updated an object in its read set, and simply means publishing the private copy.

2.3 Software transactional memory in Haskell

The concept of transactional memory presented in the preceding section is elegantly adapted in Haskell by definining two new polymorphic types: `STM a` for memory transactions, and `TVar a` for transactional variables (memory locations shared by multiple threads and updatable through transactions). Haskell's type system separates transactional variables from other data types and guarantees that they can only be accessed within transactions protected with the function `atomically`.

An *STM action* is like an I/O action in that it can have side effects, but the range of possible side effects is limited: they can't have consequences that can't be undone (such as printing something for example), therefore the only kind of possible side effect is to access transactional variables. Apart from those accesses, only pure computations can be performed inside STM actions.

Throughout the rest of this document, a transaction will therefore mean an I/O action executing an STM action atomically. STM actions execute tentatively and `atomically` exposes the result to the rest of the system if it has been run on a consistent system.

Such an action display two characteristics expected of transactions: isolation and atomicity. Furthermore, STM actions can be composed into bigger STM actions using `do`, `retry` and `OrElse` constructs, and if two STM actions working correctly are so composed, the resulting STM action also works correctly.

2.3.1 Access to shared variables

Throughout the rest of this document, a shared variable will mean a TVar and we will assume no MVar is used.

The operation `newTVar` :: $\texttt{a} -> \texttt{STM}(\texttt{TVar a})$ takes a value of type `a` and creates a new `TVar` with this value, `readTVar` :: $\texttt{TVar a} -> \texttt{STM a}$ takes a `TVar` of type `a` and returns an STM action reading its value and `writeTVar` :: $\texttt{TVar a} -> \texttt{a} -> \texttt{STM}()$ takes a `TVar` of type `a` and a value of type `a` and returns an STM action writing the `TVar`.

2.3.2 Atomically

STM actions record their accesses to `TVars` in personnal logs and perform their calculations unseen by the other STM actions. In order to make the result visible to all other STM actions, a function `atomically :: STM a -> IO a` is invoked. The function takes an STM action and delivers an I/O action that, when performed, runs the STM action atomically as a transaction: the STM interface checks that no concurrent transaction has committed conflicting updates on any of the `TVars` used by the transaction, and if no conflict has occured, the transaction commits, otherwise no modification is performed by the transaction and it is reexecuted with a fresh log.

2.3.3 Do, Retry and OrElse

STM actions are actions and can be composed using `do` as I/O actions.

The `retry :: STM a` function aborts the current STM action, rolls back and retries it.

The `OrElse :: STM a -> STM a -> STM a` function allows the programmer to define an alternative STM action to perform if an STM action retries.

2.3.4 Relations between I/O and STM

I/O actions and STM actions are strictly separated: it is not possible to perform I/O or manipulate `MVar` inside an STM action. The article [9] explains that strictly separating STM actions and I/O actions provides valuable security guarantees: as I/O actions cannot be performed inside a memory transaction, it is always possible to roll back a transaction and be certain it hasn't had any consequences.

However, it is sometimes necessary to combine I/O actions and STM actions. There is both safe and unsafe ways to do so.

If there is a need to perform I/O actions as a result of a decision made inside an atomic block, the decision may be made in a transaction and atomically returns it to the I/O caller to tell him what to do. The atomic block will then remain safe to roll back.

If it is necessary to perform an I/O action within an STM action, the function `UnsafeIoToSTM :: IO a -> STM a` makes it possible, but if a user chooses to use that possibility, the type system can't help him to make sure that all the actions performed inside the transaction can be rolled back, so it should be avoided as much as possible. Throughout the rest of this document, we will assume that this possibility is never used.

Chapter 3

Concurrency-related issues in STM Haskell

As explained in chapter 2, STM has been created to help guarantee correctness in concurrent programs without critical section usage. Whereas it does provide a sound way to do so, it does no more solve all concurrency-related issues than critical section. Some of those issues are presented in the present chapter. Precisely, the first two sections briefly discuss types of concurrency errors this document does not discuss further, while the last two sections present the kind of errors the Moth tool, presented in the next part, attempts to detect in programs: high-level data races and stale-value errors.

3.1 Termination issues

This paper exclusively concerns itself with issues regarding partial correctness and therefore does not discuss termination-affecting issues.

However, according to [4], code that uses STM will not deadlock (which would be yet another advantage of software transactional memory over lock-based approaches) but livelocks are possible if all concurrent threads simultaneously decide to abort and retry.

A *livelock* is similar to a deadlock except that, instead of passively waiting for another thread to progress, a thread takes some action to ensure progress. But it does so in such a such a way that, while none of the threads is inactive, no progress is actually made.

3.2 Low-level data races

Since no STM actions can be performed outside a transaction, a programmer cannot access a TVar without the protection of `atomically`. The occurence of low-level data race on TVar in STM Haskell programs is therefore excluded.

However, usage of software transactional memory does not eliminate subtler concurrency-related errors. If a programmer uses two separate transactions when he should have used one, he may get a high-level data race or a stale-value error.

3.3 Stale-value errors

A *stale-value* is an old value of a variable that is used after the variable has been updated.

When such an outdated value is used by a piece of code under the assumption that it is still synchronized, a *stale-value error* may occur.

Copies of `TVar` are not updated when the `TVar` is, so a copy can end up being inconsistent with the global memory state. For example, a stale-value error could be caused by using a local copy (`t`) of a TVar `x` to update another variable (`e`) if the programmer intended that `e` should be divided by 2 if it was equal to `x`:

Example 1

```
do t ← atomically (readTVar x)
   e ← if e == t then t/2 else t
```

That error could have been avoided simply by making a single transaction out of the two operations.

3.3.1 Stale-value errors definitions in the literature

Article [10] states that a value is stale if it is used outside the critical section it was defined in.

In [11], a stale-value is defined as a value of a register that originated from a different monitor block than where it is used. That definition is exactly the same that the one in [10] even if it uses a slightly different vocabulary.

In [2], a stale-value is defined as a variable replica that no longer reflects the true value of that variable.

Only the definition in [2] is a definition instead of a criterion to detect stale copy in a specific computational paradigm. However, all those definitions are sufficiently alike to show there seems to be a general agreement on the notion of stale-value.

Article [10] states that a *stale-value error* is, like a low-level data race, a scenario where a thread u reads from a shared variable l, then uses the value read, while in the meantime another thread, u', has updated l. That definition conveys an essential element lacking in most stale-value definitions: a stale-value error does not require an outdated copy: it may occur every time a piece of code uses an outdated value.

It is worth remembering, however, that a stale-value can only cause a stale-value error if the program depended in any way on the value being up to date. So, while a stale value error is such a scenario, not all such scenarios are stale-value errors.

Stale-value errors are therefore hard to detect in an automatic way since it is really the programmer intent that will distinguish between an error and a legitimate usage instead of an objective criterion.

3.4 High-level data races

A *high-level data race* is a data race occuring within a set of related variables instead of an unique variable.

An high level data race could be caused by updating a TVar `max` holding the maximal value of a list and a TVar `posmax` holding the position of that maximal value in separate transactions.

Example 2

```
do atomically (writeTVar max xs!!n)
   atomically (writeTVar posmax n)
```

The high-level data race may occur, for example, if a concurrent thread reads from `max` and `posmax` between both atomic writes and observes a situation where `posmax` does not hold the position of the value in `max`.

As with the stale-value error example 1, that error could have been avoided simply by making a single transaction out of the two.

3.4.1 High-level data races definitions in the literature

High-level data races were initially defined as follows in [5]: "A high-level data race can occur when two concurrent threads access a set V of shared variables, which should

be accessed atomically, but at least one of the threads accesses V partially several times such that those partial accesses diverge."[1].

The notion is defined more intuitively in [11] by stating that "the notion of high-level data races refers to sequences in a program where each access to shared data is protected by a lock, but the program still behaves incorrectly because operations that should be carried out atomically can be interleaved with conflicting operations"[2].

In [10], it is stated that a high-level data race is a scenario where either a thread reads separately two related variables and gets an incoherent view because, between the two reads, another thread has updated the related variables (or only one of them) or a thead writes two related variables separately and another thread reading the related variables between the two writes, may observe an incoherent memory state.

In [2], a high-level data race is defined as a violation of data consistency occuring when two synchronized blocks the programmer intended to run atomically have other code blocks interleaved between them because he believed it was sufficient to ensure their individual atomicity.

All four definitions are consistent if not identical, so there also seems to be a consensus in the literature regarding the type of error "high-level data race" refers to.

However, the definitions in [11] and [2] are not testable without further formalization, while the other two are less intuitive and may not cover as much cases but may directly be used for testing.

As with stale-value errors, high-level data races are hard to detect in an automatic way since it is the programmer's view of the relationship between variables that distinguishes between an error and a legitimate usage.

[1]Cyrille Artho, Klaus Havelund and Armin Biere, "High-Level Data Races" in *Software Testing, Verification and Reliability: Special Issue: VVEIS 2003 Workshop on Verification and Validation of Enterprise Information Systems, 13(4)*, December 2008, page 8, retrieved October 2012 from http://staff.aist.go.jp/c.artho/papers/stvr03.pdf [5].

[2]Cyrille Artho, Klaus Havelund and Armin Biere, "Using block-local atomicity to detect stale value concurrency errors" in *Proc. ATVA '04*,2004, retrieved November 2012 from ti.arc.nasa.gov/m/pub-archive/885h/0885%20(Artho).pdfstaff.aist.go.jp/c.artho/papers/vveis03.pdf [11].

Part II

Consistency verification for Haskell Transactional Memory Programs using Moth

Chapter 4

View generation

As seen in chapter 3, STM programs are not free of concurrency errors. Hence algorithms that check software transactional memory-based programs for them are necessary.

The tool presented in [2], Moth, answers that need for programs writen in Java and the current part attempts to adapt it to STM Haskell programs. Ideas presented here are therefore integrally taken from [2] but slighly adapted to the specificities of STM Haskell.

The Moth tool unifies checkers targeting different kinds of concurrency errors by providing an analysis which generates a common core of information about memory access, on the basis of which plugged-in algorithms (each detecting a specific error pattern) work.

The current chapter presents the generation of the set of memory accesses in Moth, the *view set*, and the theory behind it. The first section defines the view set, the second one presents the rules used by Moth for its generation and the last one shows an example of application of those rules on a subset of Haskell.

The next chapter will then present the plugged-in algorithms and how they use that view set.

4.1 View set

4.1.1 Views

A view expresses which TVars are accessed inside an atomic STM action (or *transaction*).

Access

Let $\mathcal{F}$ be the set of all TVar in a program and $\mathcal{A}$ be the set of all read and write accesses to variables of $\mathcal{F}$ inside transactions. An *access* $a \in \mathcal{A}$ is a triple (α, γ, β) where $\alpha = r$ if a is a read access and $\alpha = w$ if it is a write access, $\gamma \in \mathcal{F}$ is the accessed TVar and β helps keeping a "use-define" relation for each accessed TVar as follows:

If $\alpha = r$ and the value of γ will later be overwritten inside the same transaction then $\beta = \bullet$;
If $\alpha = r$ and the value of γ will not be overwritten later inside the same transaction then $\beta = \circ$;
If $\alpha = w$ and the value of γ was read before in the same transaction then $\beta = \bullet$;
If $\alpha = w$ and the value of γ was not read before in the same transaction then $\beta = \circ$;

For example, $(r, x, \bullet)$ is a read access to the variable x inside a block in which x will be written after it has been read.

In example 1 (presented on page 25), $\mathcal{F} = \{x\}$ and $\mathcal{A} = \{(r, x, \circ)\}$.

In example 2 (presented on page 26), $\mathcal{F} = \{max, posmax\}$ and $\mathcal{A} = \{(w, max, \circ), (w, posmax, \circ)\}$.

View

A view is a set of memory accesses made in a specific transaction. Formally, a *view* $v \subseteq \mathcal{A}$ of a transaction is a subset of $\mathcal{A}$ and includes all the variable accesses made inside that transaction. The set of all the views is denoted by $\mathcal{V}$.

In example 2, $v = \{(w, max, \circ)\}$ for the first transaction and $v = \{(w, posmax, \circ)\}$ for the second.

In example 3, $v = \{(w, max, \circ), (w, posmax, \circ)\}$:

Example 3

```
atomically (do writeTVar max xs!!n
               writeTVar posmax n)
```

4.1.2 Set of views for a thread

The set of views of a thread characterizes the way it may interact with other threads through shared TVars. Therefore, to detect potential concurency errors, sets of views for each thread must be generated. Furthermore, since only specific interleaving of

read and write may cause potential errors, read and write views need to be further distinguished.

Let $\mathcal{B}$ be the set of transactions, $\Gamma : \mathcal{V} \to \mathcal{B}$ be a relation returning the transaction characterized by a given view and $executes(t,b)$ be a boolean that is true when a thread t executes a transaction b. The *set of generated views of a thread t*, $V(t)$, is the set of views of each transaction executed by t: $v \in V(t) \Leftrightarrow b = \Gamma(v) \wedge executes(t,b)$.

In example 4, the set of views of the first thread is $V(t_1) = \{(w, max, \circ)\}$, that of the second $V(t_2) = \{(w, posmax, \circ)\}$ and the set of views of the last thread is $V(t_3) = \{(r, posmax, \circ), (r, max, \circ)\}$.

Example 4

```
main = do forkIO    atomically (writeTVar max xs!!n)
          forkIO    atomically (writeTVar posmax n)
          forkIO    atomically (do readTVar p posmax
                                   readTVar m max)
```

Since it will be necessary to distinguish read and write accesses, the *set of read views of a thread t* is denoted $V_r(t)$: $V_r(t) = \{(r,\gamma,\beta) | (r,\gamma,\beta) \in V(t)\}$ and the *set of write views of a thread t* is denoted $V_w(t)$: $V_w(t) = \{(w,\gamma,\beta) | (w,\gamma,\beta) \in V(t)\}$.

Continuing example 4, $V_r(t_1) = \emptyset$, $V_w(t_1) = \{(w, max, \circ)\}$, $V_r(t_2) = \emptyset$, $V_w(t_2) = \{(w, posmax, \circ)\}$, $V_r(t_3) = \{(r, posmax, \circ), (r, max, \circ)\}$ and $V_w(t_3) = \emptyset$.

4.2 View generation in Moth

The Moth tool generates the set of views of every thread in a program using static analysis and runs different concurrency errors-detecting algorithms (called *sensors*) on those views.

The view generation produces views for each transaction by starting with empty views and adding successively each access to TVar made in the action body.

For a given transaction :

1. The view generation starts with an empty view;
2. Each time an access is made to a TVar γ, that access is added to the view. Inserting a memory access (α,γ,β) into a view is defined as follows: $add : \mathcal{A} \times \mathcal{V} \to \mathcal{V}$:

$$add((\alpha,\gamma,\beta), v) \equiv \begin{cases} v \setminus \{(\alpha,\gamma,\delta)\} \cup \{(\alpha,\gamma,\beta)\} \text{ if } \alpha = r \wedge (\alpha,\gamma,\delta) \in v \wedge \delta \neq \beta \\ v \setminus \{(r,\gamma,\delta)\} \cup \{(r,\gamma,\bullet)\} \cup \{(\alpha,\gamma,\bullet)\} \text{ if } \alpha = w \wedge (r,\gamma,\delta) \in v \\ v \cup \{(\alpha,\gamma,\beta)\} \text{ otherwise} \end{cases}$$

If a transaction contains multiple accesses to a TVar, previously inserted accesses may need to be updated to reflect informations that weren't known when the access was inserted. For example, $(r,\gamma,\circ)$ will be transformed into $(r,\gamma,\bullet)$ when adding $(w,\gamma,\bullet)$.

3. Each time an STM action is called inside the body of another STM action, the callee's views, v_p, are computed in isolation then merged with the caller's, v.

   Merging views is defined as follow: $merge_P : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$: $merge_P(v_p, v) = v''$ where

   $v'' = merge(\{(r,\gamma,\beta) | (r,\gamma,\beta) \in v_p\}, v')$,
   $v' = merge(\{(w,\gamma,\beta) | (w,\gamma,\beta) \in v_p\}, v)$ and
   $merge(v_P, v) \equiv \begin{cases} v \text{ if } v_p = \emptyset \\ merge(v_p \setminus \{a\}, add(a,v)) \text{ if } \exists a \in v_p. \end{cases}$

If an STM action contains a call to another STM action, the view corresponding to the action called will be computed separetly, using the actual parameters instead of the formal parameters. The access in that view will then be added to those already computed for the calling STM action as if they were made in that STM action. Merging view is thus simply uniting two sets. The authors of [2] insist that all the write accesses should be merged before the read accesses but the procedure would work either way.

For example, if an STM action `transfer` :: `Integer` $->$ `Account` $->$ `Account` $->$ `STM`() is defined using two STM actions, `credit` :: `Integer` $->$ `Account` $->$ `STM`() and `debit` :: `Integer` $->$ `Account` $->$ `STM`() as in example 5, the views of `credit` and `debit` will be computed then added to the access already included in the view of `transfer`.

Example 5

```
transfer amount from to = do      fromVal < −readTVarfrom
                                  if(fromVal − amount) >= 0 then do
                                      debit amount from
                                      credit amount to
                                  else retry
```

4. Each time a choice instruction is used, the accesses occuring in the choice condition and in both sides of the alternative are added to the view.

At the end of the view generation algorithm, the information used by the sensors presented in the next chapter is fully computed.

4.3 Example of view generation for a subset of Haskell

In the following example, the impact of the langage constructs in a small Haskell-like language on the set of views is illustrated. It aims to illustrate how the rules defined in section 4.2 translate into Haskell.

For the purpose of this illustration, the syntax is restricted to a subset of Haskell in which neither the arguments of the functions nor the choice conditions can contain TVar and STM actions can't be nested but the view generation procedure itself is not subject to those limitations.

The fact that a set of views $\langle v \rangle$ becomes $\langle v' \rangle$ after an instruction S will be noted $\langle v, S \rangle \Rightarrow \langle v' \rangle$.

4.3.1 Language definition

In the small language considered, an instruction (S) may take three principal forms: either S is a call to a pure function or it is a call to an action. In that second case, it may either be a transaction or a "regular" IO action.

$$
\begin{array}{lll}
\texttt{S} & ::= & f\overrightarrow{\texttt{x}'} \\
 & & \text{where } \texttt{f} :: \overrightarrow{a} -> \texttt{b} \\
 & & \qquad \texttt{f}\vec{\texttt{x}} = \texttt{Y} \\
 & & \mid \texttt{f}\overrightarrow{\texttt{x}'} \\
 & & \text{where } \texttt{f} :: \overrightarrow{a} -> \texttt{IO b} \\
 & & \qquad \texttt{f}\vec{\texttt{x}} = \texttt{Z} \\
 & & \mid \texttt{atomically}(\texttt{f}\overrightarrow{\texttt{x}'}) \\
 & & \text{where } \texttt{f} :: \overrightarrow{a} -> \texttt{STM b} \\
 & & \qquad \texttt{f}\vec{\texttt{x}} = \texttt{W}
\end{array}
$$

where Y, Z and W represent, respectively, the body of a pure function, the body of an IO action, and the body of a transaction as defined below.

A pure function is an expression evaluation, a call to another pure function or a choice between two pure functions or expressions.

$$
\begin{array}{rcl}
\texttt{Y} & ::= & \texttt{e} \text{ where } \texttt{e} \text{ is an expression} \\
& & \mid \texttt{if e then Y else Y} \\
& & \mid f\,\overrightarrow{\texttt{x}'} \\
& & \text{where } \texttt{f} :: \overrightarrow{a} -> \texttt{b} \\
& & \qquad \texttt{f}\vec{\texttt{x}} = \texttt{Y}
\end{array}
$$

An IO action may contain any instruction in the language or a sequence of such instructions.

$$
\begin{array}{rcl}
\texttt{Z} & ::= & \texttt{S} \\
& & \mid \texttt{do}\{\texttt{S};\ldots;\texttt{S}\}
\end{array}
$$

A transaction is an atomic call to a pure function or to a function writing or reading a TVar. It may also be an atomic call to a sequence of such instructions or a choice between such instructions.

$$
\begin{array}{rcl}
\texttt{W} & ::= & f\,\overrightarrow{\texttt{x}'} \\
& & \text{where } \texttt{f} :: \overrightarrow{a} -> \texttt{b} \\
& & \qquad \texttt{f}\vec{\texttt{x}} = \texttt{Y} \\
& & \mid \texttt{x} = \texttt{readTVar a} \text{ where } \texttt{a} \text{ is a TVar} \\
& & \mid \texttt{writeTVar a x} \text{ where } \texttt{a} \text{ is a TVar} \\
& & \mid \texttt{do}\{\texttt{W};\ldots;\texttt{W}\} \\
& & \mid \texttt{if e then W else W} \\
& & \mid \texttt{if e then W else retry}
\end{array}
$$

4.3.2 View generation

The fact that a set of views $\langle v \rangle$ becomes $\langle v' \rangle$ after an instruction S will be noted $\langle v, S \rangle \Rightarrow \langle v' \rangle$.

Impact of the fact that an instruction is a pure function, an IO action or a transaction

$$\begin{aligned} \mathbf{S} \quad ::= \quad & \mathbf{f}\overrightarrow{\mathbf{x'}} \\ & \textbf{where } \mathbf{f} :: \overrightarrow{\mathbf{a}} -> \mathbf{b} \\ & \qquad \mathbf{f}\overrightarrow{\mathbf{x}} = \mathbf{Y} \end{aligned}$$

$\langle v, f\vec{x} \rangle \Rightarrow \langle v \rangle$, no matter what `Y` is like, because IO actions can't be found inside pure functions. The view generation procedure therefore stop here without looking at the body (`Y`) of the function `f`.

$$\begin{aligned} \mathbf{S} \quad ::= \quad & \mathbf{f}\overrightarrow{\mathbf{x'}} \\ & \textbf{where } \mathbf{f} :: \overrightarrow{\mathbf{a}} -> \textbf{IO } \mathbf{b} \\ & \qquad \mathbf{f}\overrightarrow{\mathbf{x}} = \mathbf{Z} \end{aligned}$$

If `S` is an IO action, the view generation procedure must look at the body (`Z`) of the function for calls to transactions.

$\langle v, S \rangle \Rightarrow \langle v'' \rangle$ where the set of views of `Z` is denoted v' and $\langle v'' \rangle = merge(v, v')$ using definition of merge from 4.2. Since the arguments of `f` can't be TVars, only the accesses made in the transactions in the function body will be added to the view.

$$\begin{aligned} \mathbf{S} \quad ::= \quad & \textbf{atomically } (\mathbf{f}\overrightarrow{\mathbf{x'}}) \\ & \textbf{where } \mathbf{f} :: \overrightarrow{\mathbf{a}} -> \textbf{STM } \mathbf{b} \\ & \qquad \mathbf{f}\overrightarrow{\mathbf{x}} = \mathbf{W} \end{aligned}$$

If `S` is a transaction, the view generation procedure must compute the view of that transaction and therefore look at the body (`W`) of the function called atomically to record which TVars are accessed in it.

$\langle v, S \rangle \Rightarrow \langle v'' \rangle$ where the set of views of **W** is denoted v' and $\langle v'' \rangle = merge(v, v')$ using definition of merge from 4.2. Since the arguments of `f` can't be TVar, only the accesses made in the function body will be added to the view.

View generation procedure on the body of an IO action

$$\mathbf{Z ::= S}$$

If the body of an IO action is composed of a single instruction `S`, the set of views of `Z` is the set of views of S.

$$\mathbf{Z ::= do\{S;\ldots;S\}}$$

If the body of an IO action is composed of a sequence of instructions, the set of views of `Z` is composed of the set of views of each of those instructions.

$\langle v, Z\rangle = \langle v, do\{S_1;S_2\}\rangle \Rightarrow \langle v''\rangle$ with $\langle v, S_1\rangle \Rightarrow \langle v'\rangle$ and $\langle v', S_2\rangle \Rightarrow \langle v''\rangle$

View generation procedure on the body of a transaction

$$\begin{aligned}\mathbf{W} \quad ::= \quad & \mathbf{f}\overrightarrow{\mathbf{x}} \\ & \textbf{where } \mathbf{f} :: \overrightarrow{\mathbf{a}} -> \mathbf{b} \\ & \qquad \mathbf{f}\overrightarrow{\mathbf{x}} = \mathbf{Y}\end{aligned}$$

$\langle v, W\rangle \Rightarrow \langle v\rangle$, no matter what `Y` is like, because actions can't be found inside pure functions. The view generation procedure therefore stop here without looking at the body of the function `f`.

$$\mathbf{W} \quad ::= \quad \mathbf{x = readTVar\ a\ where\ a\ is\ a\ TVar}$$

$\langle v, W\rangle \Rightarrow \langle v'\rangle$ where $\langle v'\rangle = add((r, a, \circ), v)$ using the definitions of access and add from 4.2.

$$\mathbf{W} \quad ::= \quad \mathbf{writeTVar\ a\ x\ where\ a\ is\ a\ TVar}$$

$\langle v, W\rangle \Rightarrow \langle v'\rangle$ where $\langle v'\rangle = add((w, a, \circ), v)$ using the definitions of access and add from 4.2.

$$\mathbf{W ::= do\{W;\ldots;W\}}$$

If the body of an STM action is composed of a sequence of instructions, its set of views is composed of of the access made in each of the instructions.

$\langle v, W\rangle = \langle v, do\{W_1;W_2\}\rangle \Rightarrow \langle v''\rangle$ with $\langle v, W_1\rangle \Rightarrow \langle v'\rangle$ and $\langle v', W_2\rangle \Rightarrow \langle v''\rangle$

W ::= **if e then W else W**

If the body of a transaction is composed of a choice between instructions, the set of views of `W` is composed of the access made in each of the instructions.

$\langle v, W \rangle = \langle v, if\ e\ then\ W_1\ else\ W_2 \rangle \Rightarrow \langle v' \rangle$ with $\langle v, W_1 \rangle \Rightarrow \langle v'' \rangle$, $\langle v, W_2 \rangle \Rightarrow \langle v''' \rangle$ and $\langle v' \rangle = merge(v, v'' \cup v''')$ using definition of merge from 4.2.

W ::= **if e then W else retry**

If the body of a transaction is composed of a choice between a normal instruction and a retry instruction, the set of views of `W` is composed of the access made in the normal instruction because there won't be any additional access while retrying.

$\langle v, W \rangle = \langle v, if\ e\ then\ W_1\ else\ retry \rangle \Rightarrow \langle v' \rangle$ with $\langle v, W_1 \rangle \Rightarrow \langle v' \rangle$.

Chapter 5

Error detection

The Moth tool consists of a view generation algorithm and an extendable number of checking algorithms. The two checking algorithms included by default in the tool are a checker for stale-value errors (a modified version of the algorithm presented in [12]) and one for high-level data races (an adaptation of the approach of [5] for a transactional memory environment).

The previous chapter presented the view generation. The present chapter presents the sensors using the generated views and the theory behind them. The first section briefly discusses the Moth tool, the second presents a view-related criterion for stale-value errors and the Moth sensor using that criterion, and the last section does the same with high-level data races.

5.1 The Moth tool

The Moth tool, introduced in [2], is a piece of software that tests transactional memory-based Java program for existence of potential high-level data races and stale-value errors. It generates the set of views of every STM action in a program and runs different concurrency-errors detecting algorithms (called *sensors*) on those views.

The original tool runs only two sensors: the *view consistency sensor*, targeting high-level data races, and the *single variable sensor*, targeting stale-value errors, but Moth is built to be extensible by allowing other sensors to be plugged in.

Each plugin detects a specific kind of error, but, as the sets of conflicts detected are not necessarily disjoint, they are merged before being presented to the user to avoid multiple reporting. An advantage in terms of performance is that the sensors are completely independent from one another, so they can be executed in parallel.

In order to make automatic detection of both kinds of errors possible, formal crite-

ria for each error type must be defined. Those criteria have inherent limits: they will not be able to encompass the complexity of the consistency error they target and will, therefore, characterize some rightful usage as erroneous and fail to detect some erroneous usages.

The imperfection of the criteria and, therefore, of the tool is the inevitable price to pay for not asking the programmer to document his intent, because the tool needs to use formal rules to define what is an expected behavior and what constitutes a logical mistake, while consistency errors really depends on the programmer intent. False positive and/or false negative can consequently not be entirely avoided in tools such as Moth.

5.2 Stale-value errors detection

5.2.1 Single-variable safety property

As seen in section 3.3, a stale-value error occurs when an outdated value of a `TVar` is used under the assumption that it is still the current value.

When a `TVar` is used in several operations in a thread, the operations are often logically related. Since there is no way of knowing what the logical link is, or even whether there really is one without asking the programmer, the criterion makes the conservative assumption that operations on a `TVar` taking place in separate transactions are always related in such a way that a stale-value error will occur if another thread updates the `TVar`.

Conversely, that criterion narrows the definition of stale-value errors because it excludes their occurence outside transactions or logical relations between operations on several `TVars`.

In the context of the sensor, that hypothesis means that a variable read (and not overwritten) in a view of a thread and written in another view of the same thread may generate a stale value, which narrows the range of stale-value errors by restricting targeted errors to errors affecting a single variable.

Stale-value error

Let $\mathcal{T}$ be the set of threads of a given program. For $t \in \mathcal{T}$, $psv(\gamma, t)$ is a boolean that is true when $\gamma \in \mathcal{F}$ has a possible stale value for t. A variable $\gamma \in \mathcal{F}$ *has a possible stale value* if $\exists v_1 \neq v_2 \in V(t) : (r, \gamma, \circ) \in v_1 \wedge (w, \gamma, \beta) \in v_2$.

For example, with the following piece of code in a thread t:

Example 6

$$\texttt{oldx} \leftarrow \texttt{atomically}(\texttt{readTVar x})$$
$$\texttt{atomically}(\texttt{writeTVar x oldx} + 1),$$

the view of the first transaction, $v_1 = \{(r, \texttt{x}, \circ)\}$, and the view of the second transaction, $v_2 = \{(w, \texttt{x}, \circ)\}$, are two different views of thread t and $(r, \texttt{x}, \circ) \in v_1 \wedge (w, \texttt{x}, \circ) \in v_2$ so x *has a possible stale value* for t ($psv(\texttt{x}, t)$).

Single variable safety property

A program will be characterized as free of stale-value errors if, for any pair of threads, none of them presents a risk of possible stale value for a TVar or, if one does, the other thread never writes that TVar so that the risk will not materialize.

For $t \in \mathcal{T}$, let $writes(\gamma, t)$ be a boolean that is true if there is an access to γ in $V_w(t)$. A program is *free of stale-value errors* iff $\forall t_1 \neq t_2 \in \mathcal{T} : pSafe(t_1, t_2) \wedge pSafe(t_2, t_1)$ where $pSafe(t_1, t_2) \Leftrightarrow \forall \gamma \in \mathcal{F} : \neg writes(\gamma, t_1) \vee \neg psv(\gamma, t_2)$.

Continuing the previous example, assuming $\mathcal{F} = \{\texttt{x}\}$, the sensor will test each other thread t' in the program against t to check $pSafe(t', t)$, namely $\neg writes(\texttt{x}, t') \vee \neg psv(\texttt{x}, t) = \neg writes(\texttt{x}, t')$ since we know that x *has a possible stale value* for t. The program will therefore only be safe if no other thread writes x.

5.2.2 Single-variable sensor

The single-variable sensor is inspired by [12] so it targets successive transactions that should be merged into a single one to avoid stale-value errors, and reports a warning when the single variable safety property is violated.

The sensor analyses the views of each transaction to ensure that the program is free of stale-value errors by checking, for each pair of threads and each variable in the program, that either one of the thread does not write the variable, or the variable cannot generate a possible stale value error in the other. If none of those properties is true, the pair of transactions is reported to the programmer to suggest he makes a single transaction out of the two.

Like the original algorithm in [12], this sensor only targets specific patterns and is neither sound, nor complete (though, according to [2], experimental results seem to indicate that the implementation presented generates less false positives than the original implementation).

The main problem regarding soundness is that the definition of possible stale value does not account for whether there really is a logical link between two operations on a

variable or what that link is.

The following piece of code would cause a possible stale value, according to the criterion, despite the fact that the write in the second transaction is completely independent of the read in the preceding transaction

Example 7

```
oldx ← atomically(readTVar x)
       atomically(writeTVar x 1)
```

That problem could be solved by tracking the value read in the first transaction to see if it influences the write in the second transaction or a decision leading to it, but that analysis can't be conducted using only the views defined in the view generation procedure.

Most importantly, the sensor isn't complete, firstly because it only reports possible stale-value affecting transactional variables (an issue not present in the original tool since there was no specific type for shared variables), and secondly because it does not reports usage of a stale value to update another transactional variable. Neither of the following examples would therefore be reported by the sensor.

Example 8

```
oldx ← atomically(readTVar x)
                  newcopy ← oldx
```

Example 9

```
oldx ← atomically(readTVar x)
  atomically(writeTVar y oldx)
```

That kind of stale-value error would however be detected by the additional sensor considered in section 8.3.3.

5.3 High-level data race detection

5.3.1 View consistency

As seen in section 3.4, a high-level data race occurs when logically related variables are not accessed atomically.

Since there is no way of knowing what variables are logically related and which partial[1] access to them is legitimate without asking the programmer, the tool will make the conservative assumption that variables that are accessed atomically somewhere in the program are logically related and that divergent partial accesses to those are erroneous and must be reported.

In the context of the view consistency sensor, the view notation can be shortened by omiting the use-define relation because it is irrelevant to the identification of related variables.

Maximal view

Maximal views are the views that are not subsets of other views of the same thread. The maximal view of a thread t is denoted $M(t)$. A view $v_m \in M(t) \Leftrightarrow v_m \in V(t) \wedge (\forall v \in V(t) : v_m \subseteq v \Rightarrow v = v_m)$.

The maximal view of a thread is the set of related variables infered by the view concurrency sensor: since they are the biggest set of `TVars` accessed in a single transaction, the sensor assume they are related.

Since only specific interleaving of read and write may cause potential errors, read and write maximal views need to be further distinguished.

The read maximal view of a thread t is denoted $M_r(t)$ and its write maximal view $M_w(t)$. A view $v_m \in M_\alpha(t)$ where $\alpha \in \{r,w\} \Leftrightarrow v_m \in V_\alpha(t) \wedge (\forall v \in V_\alpha(t) : v_m \subseteq v \Rightarrow v = v_m)$.

When discussing read and write maximal views, it is not necessary to repeat the access type. Therefore, the accesses in those views are reduced to the accessed *TVar*.

Example 10

$$\begin{aligned} \text{If } t_1 = \texttt{do atomically(\ writeTVar max xs!!n)} \\ \texttt{atomically(\ writeTVar posmax n)} \\ \text{and } t_2 = \texttt{atomically(do writeTVar max xs!!n} \\ \texttt{writeTVar posmax n)}, \end{aligned}$$

The read and write maximal views of t_1 are $M_r(t_1) = \emptyset$ and $M_w(t_1) = \{\{\texttt{max}\},\{\texttt{posmax}\}\}$ and its maximal view is $M(t_1) = \{\{(\texttt{w},\texttt{max})\},\{(\texttt{w},\texttt{posmax})\}\}$. That of t_2 are $M_r(t_2) = \emptyset$ and $M_w(t_2) = \{\{\texttt{max},\texttt{posmax}\}\}$ and its maximal view is $M(t_2) = \{\{(\texttt{w},\texttt{max}),(\texttt{w},\texttt{posmax})\}\}$. The sensor would therefore assume that `max` and `posmax` are related because they appear together in the maximal view of t_2.

[1] An access to a set of variable is partial if it is an access to some variables of the set but not all of them.

Overlapping views

Overlapping views are non-empty intersections of views. The view of a thread that can cause concurrency-related errors are those that overlap with the views of other threads.

The read overlapping view of a thread t with a view of another thread, v_m is defined as follows: $overlap_r(t, v_m) = \{v_m \cap v \mid (v \in V_r(t)) \wedge (v_m \cap v \neq \emptyset)\}$.

The write overlapping view of a thread t with a view of another thread, v_m is defined as follows: $overlap_w(t, v_m) = \{v_m \cap v \mid (v \in V_w(t)) \wedge (v_m \cap v \neq \emptyset)\}$.

View compatibility

[13] defines a *chain* as a subset A of a partially ordered set $(B, <)$ which is totally ordered by $<$. Views are compatible if they form a chain of $\mathcal{V}$ using the usual set inclusion as the partial order. If views do not form a chain, the access they contain will be qualified as *divergent*.

A set of views of a thread t is *read compatible* with a maximal view v_m of another thread, which will be noted $comp_r(t, v_m)$, if and only if every read overlapping view of t with v_m form a chain.

We have $comp_r(t, v_m) \Leftrightarrow \forall v_1, v_2 \in overlap_r(t, v_m) : v_1 \subseteq v_2 \vee v_2 \subseteq v_1$.

A set of views of a thread t is *write compatible* with a maximal view v_m of another thread, which will be noted $comp_w(t, v_m)$, if and only if every write overlapping view of t with *vm* form a chain.

We have $comp_w(t, v_m) \Leftrightarrow \forall v_1, v_2 \in overlap_w(t, v_m) : v_1 \subseteq v_2 \vee v_2 \subseteq v_1$.

Continuing example 10, $overlap_r(t_1, M_r(t_2)) = \emptyset$, $overlap_w(t_1, M_w(t_2)) = \{\{\texttt{max}\}, \{\texttt{posmax}\}\}$, $overlap_r(t_2, M_r(t_1)) = \emptyset$, $overlap_w(t_2, M_w(t_1)) = \{\{\texttt{max}\}, \{\texttt{posmax}\}\}$.

Accordingly $comp_r(t_1, M(t_2))$ is true because $overlap_r(t_1, M_(t_2)) = \emptyset$ but $comp_w(t_1, M(t_2))$ is false because $\{\texttt{max}\}, \{\texttt{posmax}\} \in overlap_w(t_1, M_(t_2))$ but we have neither $\{\texttt{max}\} \subseteq \{\texttt{posmax}\}$ nor $\{\texttt{posmax}\} \subseteq \{\texttt{max}\}$

View safety property

The following property has to be verified in order to guarantee the *absence of high-level data races*: $\forall t_1 \neq t_2$, m_r is the maximal read view of thread t_1, m_w is the maximal write view of thread t_1 : $comp_w(t_2, m_r) \wedge comp_r(t_2, m_w) \wedge comp_w(t_2, m_w)$.

The view of each thread needs to be write compatible with both the read and write maximal views of every other thread and read compatible with their write maximal

view. Indeed, if a thread is not read compatible with the write maximal view of another thread, it may observe an incoherent state, and if a thread is not write compatible with the maximal view of another thread, it may cause an incoherent state.

5.3.2 View consistency sensor

The view consistency sensor is inspired by [5], thus making the same assumption about maximal views being likely candidates for sets of variables that should be accessed atomically. Therefore, the sensor reports a warning for each violation of the view safety property.

The sensor analyses the views of each transaction to insure the program is free of high-level data races by checking for each pair of threads that one of them is write compatible with both the maximal read view and the maximal write view of the other and read compatible with its maximal write view.

The main problem regarding completeness is that it requires a correct atomic access to a set of related variables to take place to be able to detect variables that are related and report incorrect usage.

The main problem regarding soundness is that any usage of the variables in a single transaction may make them appear related even if there is no logical relation between them.

Both of those problems are related to the fact of detecting related variables automatically, instead of getting the information from an exterior source. They are therefore the inevitable price of not asking the programmer to do additional work.

5.4 Conclusion

The context of STM Haskell have both assets and drawbacks for the Moth analysis.

The peculiarities of the Haskell type system imply that low-level data races don't need to be detected by using another tool before running Moth, and make the procedures a lot faster because the tool can identify and target piece of code in which transactional variables can be accessed.

However, since only `TVar` usage is tracked by the sensor, a broadest range of errors will not be reported by the sensors.

Chapter 6

Moth analysis for STM Haskell: an example

The current chapter presents an example of the way Moth works using an adaptation of the STM example in [14]. It illustrates both the view generation presented in chapter 4 and the default sensor presented in chapter 5.

The analysed program is presented below. It consists of three threads: the main thread t_0, and threads t_1 and t_2, created on line 13 and 14 respectively.

Example 11

```
1  module Main where
2
3  import Control.Concurrent (forkIO)
4  import Control.Concurrent.STM
5  import Control.Monad (forever)
6  import System.Exit (exitSuccess)
7
8  type Account = TVar Integer
9
10 main = do
11     bob <- newAccount 10000
12     jill <- newAccount 4000
13     forkIO (atomically (transfer 1 bob jill))
14     forkIO (atomically (transfer 1 bob jill))
15     forever ( do
16       bobBalance <- atomically (readTVar bob)
17       jillBalance <- atomically (readTVar jill)
```

```
18        putStrLn ("Bob's balance: " ++ show bobBalance)
19        putStrLn (", Jill's balance: " ++ show jillBalance)
20        if bobBalance == 9998 then exitSuccess
21          else putStrLn "Trying again.")
22
23  newAccount :: Integer -> IO Account
24  newAccount amount = newTVarIO amount
25
26  transfer :: Integer -> Account -> Account -> STM ()
27  transfer amount from to = do
28      fromVal <- readTVar from
29      if (fromVal - amount) >= 0
30          then do
31                  debit amount from
32                  credit amount to
33          else retry
34
35  credit :: Integer -> Account -> STM ()
36  credit amount account = do
37      current <- readTVar account
38      writeTVar account (current + amount)
39
40  debit :: Integer -> Account -> STM ()
41  debit amount account = do
42      current <- readTVar account
43      writeTVar account (current - amount)
```

6.1 View generation

The first procedure executed by Moth is the view generation which generates a common core of information about memory access in the program.

6.1.1 View generation for t_0

The code for t_0 is:

```
bob <- newAccount 10000
jill <- newAccount 4000
forever(do
  bobBalance <-atomically (readTVar bob)
  jillBalance<-atomically(readTVar jill)
  putStrLn ("Bob's balance: " ++ show bobBalance)
  putStrLn (", Jill's balance: " ++ show jillBalance)
  if bobBalance == 9998 then exitSuccess
```

```
    else putStrLn "Trying again.")
```

The thread executes two transactions.
The first transaction, `bobBalance` $<-$ `atomically(readTVar bob)`, makes a single access $(r, \texttt{bob}, \circ)$. The second, `jillBalance` $<-$ `atomically(readTVar jill)`, also makes a single access, $(r, \texttt{jill}, \circ)$.

The set of views for t_0 is composed of the view of the first transaction, v_1, and the view of the second transaction, v_2.

Applying the view generation step-by-step on v_1, we start with $v_1 = \emptyset$. We then add the single access $(r, \texttt{bob}, \circ)$ using the add definition $add((r, \texttt{bob}, \circ), \emptyset) = \emptyset \cup \{(r, \texttt{bob}, \circ)\}$ to get $v_1 = \{(r, \texttt{bob}, \circ)\}$.

Similarly, we get $v_2 = \{(r, \texttt{jill}, \circ)\}$.

Finally, $V(t_0) = \{\{(r, \texttt{bob}, \circ)\}, \{(r, \texttt{jill}, \circ)\}\}$ with set of write views $V_w(t_0) = \emptyset$ and set of read views $V_r(t_0) = \{\{(r, \texttt{bob}, \circ)\}, \{(r, \texttt{jill}, \circ)\}\}$.

6.1.2 View generation for t_1

Intuitive idea

Using the definition of the functions transfer, credit and debit, the code for t_1 could be rewritten as follows:

```
atomically (
    fromVal <- readTVar bob
    if (fromVal - 1) >= 0
        then do
               current <- readTVar bob
               writeTVar bob (current - 1)
               current <- readTVar jill
               writeTVar jill (current + 1)
        else retry)
```

Intuitively, the access made in the single transaction executed by the thread are $(r, \texttt{bob}, \bullet)$,$(w, \texttt{bob}, \bullet)$, $(r, \texttt{jill}, \bullet)$ and $(w, \texttt{jill}, \bullet)$ and the set of views for t_1 is $V(t_1) = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (r, \texttt{jill}, \bullet), (w, \texttt{jill}, \bullet)\}$.

View generation for t_1

The code for t_1 is `atomically (transfer 1 bob jill)`. Applying the view generation algorithm to `atomically(transfer 1 bob jill)`, we start with $v = \emptyset$.

Since there is a call to `transfer`, we need to compute v', the view of `transfer 1 bob jill`, then merge it with $\emptyset$ to get the new value of v. The value of v will therefore only be computed in paragraph "call resolution for the call to transfer" on page 52, after computing v' in isolation.

$$\{v = \emptyset\}$$
$$\texttt{atomically}(\texttt{transfer 1 bob jill})$$
$$\{v = merge(\emptyset, v') = v'\}$$

View generation for the call to transfer

v' is the view of

```
do fromVal <- readTVar bob
    if (fromVal - 1) >= 0
        then do
               debit 1 bob
               credit 1 jill
        else retry
```

We start with $v' = \emptyset$ then we add the access $(r, \texttt{bob}, \circ)$ using the add definition $add((r, \texttt{bob}, \circ), \emptyset) = \emptyset \cup \{(r, \texttt{bob}, \circ)\}$ to get $v' = \{(r, \texttt{bob}, \circ)\}$.

Since the next instruction is a choice instruction, we need to add both the access in the "then" part and the "else" part, but, since the "else" part is a retry, it won't add any new access and we can focus on the "then" part.

We therefore need to compute v'', the view of the call `debit 1 bob`, then merge it with $\{(r, \texttt{bob}, \circ)\}$ to get the updated value of v'. The value of v' after the call to `debit` will consequently only be computed in paragraph "call resolution for the call to debit" on page 49, after computing v'' in isolation.

We will then need to compute v''', the view of the call `credit 1 jill`, and merge it with the updated value of v' to get the final value of v'. The final value of v' will thus only be computed in paragraph "call resolution for the call to credit" on page 51, after computing v''' in isolation.

$\{v' = \emptyset\}$
`do fromVal` $<-$ `readTVar bob`
$\{v' = \emptyset \cup \{(r, \texttt{bob}, \circ)\} = \{(r, \texttt{bob}, \circ)\}\}$
`if(fromVal` $- 1) >= 0$ `then do`
`debit 1 bob`
$\{v' = merge(v'', \{(r, \texttt{bob}, \circ)\})\}$ *where* v'' *is the view of* `debit`
`credit 1 jill`
$\{v' = merge(v''', merge(v'', \{(r, \texttt{bob}, \circ)\}))\}$ *where* v''' *is the view of* `credit`
`else retry`
$\{v' = merge(v''', merge(v'', \{(r, \texttt{bob}, \circ)\}))\}$

View generation for the call to debit

v'' is the view of

```
do current <- readTVar bob
   writeTVar bob (current - 1)
```

Using the view generation algorithm step-by-step, we start with $v'' = \emptyset$. We then add the access $(r, \texttt{bob}, \circ)$ using the add definition $add((r, \texttt{bob}, \circ), \emptyset) = \emptyset \cup \{(r, \texttt{bob}, \circ)\}$ to get $v'' = \{(r, \texttt{bob}, \circ)\}$.

Since $(r, \texttt{bob}, \circ) \in v''$ and we now need to add the access $(w, \texttt{bob}, \bullet)$. We use the add definition, $add((w, \texttt{bob}, \bullet), \{(r, \texttt{bob}, \circ)\}) = \{(r, \texttt{bob}, \circ)\} \setminus \{(r, \texttt{bob}, \circ)\} \cup \{(r, \texttt{bob}, \bullet)\} \cup \{(w, \texttt{bob}, \bullet)\}$ to get $v'' = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}$.

$\{v'' = \emptyset\}$
`do current` $<-$ `readTVar bob`
$\{v'' = \emptyset \cup \{(r, \texttt{bob}, \circ)\} = \{(r, \texttt{bob}, \circ)\}$
`writeTVar bob (current` $- 1$`)`
$\{v'' = \{(r, \texttt{bob}, \circ)\} \setminus \{(r, \texttt{bob}, \circ)\} \cup \{(r, \texttt{bob}, \bullet)\} \cup \{(w, \texttt{bob}, \bullet)\} = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}\}$

Call resolution for the call to debit

We need to merge v'' with $\{(r, \texttt{bob}, \circ)\}$ to get the new value of v':

$$
\begin{array}{lcl}
merge(v'', \{(r, \texttt{bob}, \circ)\}) & = & merge(\{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}, \{(r, \texttt{bob}, \circ)\}) \\
& = & merge(\{(r, \texttt{bob}, \bullet)\}, merge(\{(w, \texttt{bob}, \bullet)\}, \{(r, \texttt{bob}, \circ)\})) \\
& = & merge(\{(r, \texttt{bob}, \bullet)\}, merge(\emptyset, add((w, \texttt{bob}, \bullet), \{(r, \texttt{bob}, \circ)\}))) \\
& = & merge(\{(r, \texttt{bob}, \bullet)\}, merge(\emptyset, \{(r, \texttt{bob}, \circ)\} \setminus \{(r, \texttt{bob}, \circ)\} \cup \{(r, \texttt{bob}, \bullet)\} \cup \{(w, \texttt{bob}, \bullet)\})) \\
& = & merge(\{(r, \texttt{bob}, \bullet)\}, merge(\emptyset, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\})) \\
& = & merge(\{(r, \texttt{bob}, \bullet)\}, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}) \\
& = & merge(\emptyset, add((r, \texttt{bob}, \bullet), \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\})) \\
& = & merge(\emptyset, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\} \cup \{(r, \texttt{bob}, \bullet)\}) \\
& = & merge(\emptyset, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}) \\
& = & \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}
\end{array}
$$

Accordingly,

$\{v' = \emptyset\}$
`do fromVal` $<$ `-readTVar bob`
$\{v' = \emptyset \cup \{(r, \texttt{bob}, \circ)\} = \{(r, \texttt{bob}, \circ)\}\}$
`if(fromVal` $-$ `1)` $>=$ `0 then do`
`debit 1 bob`
$\{v' = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}\}$
`credit 1 jill`
$\{v' = merge(v''', \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\})\}$
`else retry`
$\{v' = merge(v''', \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\})\}$

View generation for the call to credit

v''' is the view of

```
do current <- readTVar jill
   writeTVar jill (current + 1)
```

Using the view generation algorithm step-by-step, we start with $v''' = \emptyset$. We then add the access $(r, \texttt{jill}, \circ)$ using the add definition $add((r, \texttt{jill}, \circ), \emptyset) = \emptyset \cup \{(r, \texttt{jill}, \circ)\}$ to get $v''' = \{(r, \texttt{jill}, \circ)\}$.

Since $(r, \texttt{jill}, \circ) \in v'''$ and we now need to add the access $(w, \texttt{jill}, \bullet)$ using the add definition $add((w, \texttt{jill}, \bullet), \{(r, \texttt{jill}, \circ)\}) = \{(r, \texttt{jill}, \circ)\} \setminus \{(r, \texttt{jill}, \circ)\} \cup \{(r, \texttt{jill}, \bullet)\} \cup \{(w, \texttt{jill}, \bullet)\}$ to get $v''' = \{(r, \texttt{jill}, \bullet), (w, \texttt{jill}, \bullet)\}$.

$\{v''' = \emptyset\}$
$\texttt{do current} < -\texttt{readTVar jill}$
$\{v''' = \emptyset \cup \{(r, \texttt{jill}, \circ)\} = \{(r, \texttt{jill}, \circ)\}\}$
$\texttt{writeTVar jill (current} + 1\texttt{)}$
$\{v''' = \{(r, \texttt{jill}, \circ)\} \setminus \{(r, \texttt{jill}, \circ)\} \cup \{(r, \texttt{jill}, \bullet)\} \cup \{(w, \texttt{jill}, \bullet)\} = \{(r, \texttt{jill}, \bullet), (w, \texttt{jill}, \bullet)\}\}$

Call resolution for the call to credit

Carrying on with $v' = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}$, we then need to merge it with v''' to get the final value of v':

$$
\begin{aligned}
merge(v''', \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}) &= merge(\{(r, \texttt{jill}, \bullet), (w, \texttt{jill}, \bullet)\}, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}) \\
&= merge(\{(r, \texttt{jill}, \bullet)\}, merge(\{(w, \texttt{jill}, \bullet)\}, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\})) \\
&= merge(\{(r, \texttt{jill}, \bullet)\}, merge(\emptyset, add((w, \texttt{jill}, \bullet), \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}))) \\
&= merge(\{(r, \texttt{jill}, \bullet)\}, merge(\emptyset, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\} \cup \{(w, \texttt{jill}, \bullet)\})) \\
&= merge(\{(r, \texttt{jill}, \bullet)\}, merge(\emptyset, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet)\})) \\
&= merge(\{(r, \texttt{jill}, \bullet)\}, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet)\}) \\
&= merge(\emptyset, add((r, \texttt{jill}, \bullet), \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet)\})) \\
&= merge(\emptyset, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet)\} \cup \{(r, \texttt{jill}, \bullet)\}) \\
&= merge(\emptyset, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet), (r, \texttt{jill}, \bullet)\}) \\
&= \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet), (r, \texttt{jill}, \bullet)\}
\end{aligned}
$$

Accordingly,

$\{v' = \emptyset\}$
`do fromVal` $<$ `−readTVar bob`
$\{v' = \emptyset \cup \{(r, \texttt{bob}, \circ)\} = \{(r, \texttt{bob}, \circ)\}\}$
`if(fromVal − 1)` $>=$ `0 then do`
`debit 1 bob`
$\{v' = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet)\}\}$
`credit 1 jill`
$\{v' = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet), (r, \texttt{jill}, \bullet)\}\}$
`else retry`
$\{v' = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet), (r, \texttt{jill}, \bullet)\}\}$

Call resolution for the call to transfer

Using $v' = \{(r, \texttt{jbob}, \bullet), (w, \texttt{bob}, \bullet), (r, \texttt{jill}, \bullet), (w, \texttt{jill}, \bullet)\}$ and merging it with $\emptyset$,
we get $v = \mathit{merge}(\emptyset, \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet), (r, \texttt{jill}, \bullet)\})$
$= \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet), (r, \texttt{jill}, \bullet)\}$.

The intuition was correct: the step-by-step application of the view generation algorithm shows that the set of views for t_1 is $V(t_1) = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (r, \texttt{jill}, \bullet), (w, \texttt{jill}, \bullet)\}$ with read and write views $V_r(t_1) = \{(r, \texttt{bob}, \bullet), (r, \texttt{jill}, \bullet)\}$ and $V_w(t_1) = \{(w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet)\}$.

6.1.3 View generation for t_2

The code for t_2 is identical to that of t_1.

We have therefore $V(t_2) = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (r, \texttt{jill}, \bullet), (w, \texttt{jill}, \bullet)\}$, $V_r(t_2) = \{(r, \texttt{bob}, \bullet), (r, \texttt{jill}, \bullet)\}$ and $V_w(t_2) = \{(w, \texttt{bob}, \bullet), (w, \texttt{jill}, \bullet)\}$

6.2 Error detection

Using the view generated in section 6.1, Moth uses the sensors to detect potential concurrency errors.

As a reminder, the view of the three threads in the program are:

$V(t_0)$	$\{\{(r,\texttt{bob},\circ)\},\{(r,\texttt{jill},\circ)\}\}$
$V_r(t_0)$	$\{\{(r,\texttt{bob},\circ)\},\{(r,\texttt{jill},\circ)\}\}$
$V_w(t_0)$	$\emptyset$
$V(t_1)$	$\{(r,\texttt{bob},\bullet),(w,\texttt{bob},\bullet),(r,\texttt{jill},\bullet),(w,\texttt{jill},\bullet)\}$
$V_r(t_1)$	$\{(r,\texttt{bob},\bullet),(r,\texttt{jill},\bullet)\}$
$V_w(t_1)$	$\{(w,\texttt{bob},\bullet),(w,\texttt{jill},\bullet)\}$
$V(t_2)$	$\{(r,\texttt{bob},\bullet),(w,\texttt{bob},\bullet),(r,\texttt{jill},\bullet),(w,\texttt{jill},\bullet)\}$
$V_r(t_2)$	$\{(r,\texttt{bob},\bullet),(r,\texttt{jill},\bullet)\}$
$V_w(t_2)$	$\{(w,\texttt{bob},\bullet),(w,\texttt{jill},\bullet)\}$

6.2.1 Single-variable sensor

The sensor attempts to detect related operations on a `TVar` taking place in separate transactions by testing the single variable safety property.

The single-variable sensor assumes that a couple of thread is safe ($psafe$) for a TVar γ iff one of them does not read γ ($\neg writes$) or the other can not cause a stale value error for γ ($\neg psv$) (because it does not read and update γ in separate transactions).

To check the single variable safety property for the program, we must have

$$[psafe(t_0,t_1) \wedge psafe(t_1,t_0)] \bigwedge [psafe(t_0,t_2) \wedge psafe(t_2,t_0)] \bigwedge [psafe(t_1,t_2) \wedge psafe(t_2,t_1)] \quad = \quad true$$

which is equivalent to

$$\begin{cases} psafe(t_0,t_1) & = true \\ psafe(t_1,t_0) & = true \\ psafe(t_0,t_2) & = true \\ psafe(t_2,t_0) & = true \\ psafe(t_1,t_2) & = true \\ psafe(t_2,t_1) & = true \end{cases}$$

Namely,

$$\begin{cases} (\forall \gamma \in \mathcal{F} : \neg writes(\gamma, t_0) \vee \neg psv(\gamma, t_1)) & = true \\ (\forall \gamma \in \mathcal{F} : \neg writes(\gamma, t_1) \vee \neg psv(\gamma, t_0)) & = true \\ (\forall \gamma \in \mathcal{F} : \neg writes(\gamma, t_0) \vee \neg psv(\gamma, t_2)) & = true \\ (\forall \gamma \in \mathcal{F} : \neg writes(\gamma, t_2) \vee \neg psv(\gamma, t_0)) & = true \\ (\forall \gamma \in \mathcal{F} : \neg writes(\gamma, t_1) \vee \neg psv(\gamma, t_2)) & = true \\ (\forall \gamma \in \mathcal{F} : \neg writes(\gamma, t_2) \vee \neg psv(\gamma, t_1)) & = true \end{cases}$$

where $\mathcal{F} = \{\texttt{bob}, \texttt{jill}\}$.

Since there is only one view for both t_1 and t_2, $\neg psv(\gamma, t_1)$ and $\neg psv(\gamma, t_2))$ are both trivially true $\forall \gamma \in \mathcal{F}$, and the previous system becomes:

$$\begin{cases} true & = true \\ (\forall \gamma \in \mathcal{F} : \neg writes(\gamma, t_1) \vee \neg psv(\gamma, t_0)) & = true \\ true & = true \\ (\forall \gamma \in \mathcal{F} : \neg writes(\gamma, t_2) \vee \neg psv(\gamma, t_0)) & = true \\ true & = true \\ true & = true \end{cases}$$

Using the value of $\mathcal{F}$, we get:

$$\begin{cases} true & = true \\ (\neg writes(\texttt{bob}, t_1) \vee \neg psv(\texttt{bob}, t_0)) \wedge (\neg writes(\texttt{jill}, t_1) \vee \neg psv(\texttt{jill}, t_0)) & = true \\ true & = true \\ (\neg writes(\texttt{bob}, t_2) \vee \neg psv(\texttt{bob}, t_0)) \wedge (\neg writes(\texttt{jill}, t_2) \vee \neg psv(\texttt{jill}, t_0)) & = true \\ true & = true \\ true & = true \end{cases}$$

Since $V(t_1) = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (r, \texttt{jill}, \bullet), (w, \texttt{jill}, \bullet)\}$
and $V(t_2) = \{(r, \texttt{bob}, \bullet), (w, \texttt{bob}, \bullet), (r, \texttt{jill}, \bullet), (w, \texttt{jill}, \bullet)\}$, the previous system becomes:

$$\begin{cases} true & = true \\ (false \vee \neg psv(\texttt{bob}, t_0)) \wedge (false \vee \neg psv(\texttt{jill}, t_0)) & = true \\ true & = true \\ (false \vee \neg psv(\texttt{bob}, t_0)) \wedge (false \vee \neg psv(\texttt{jill}, t_0)) & = true \\ true & = true \\ true & = true \end{cases}$$

The single variable safety property is therefore reduced to $\neg psv(\texttt{bob}, t_0) \wedge \neg psv(\texttt{jill}, t_0) = true$, which is true since neither *bob*, nor *jill*, appear in both views of t_0.

Hence, the single variable safety property is verified for the program in example 11, and the single-variable sensor won't report a warning.

6.2.2 View consistency sensor

The sensor attempts to detect sets of related `TVars` and to detect problematic cases of partial access to variables of such sets by testing the view consistency property.

Two threads are deemed compatible by the view consistency sensor if each of them is read compatible ($comp_r$) with the write maximal view of the others and write compatible ($comp_w$) with both the read and write maximal view of the other.

As a reminder, $overlap_r(t,v)$ is the non-empty intersection of the read view of a thread t with another view sv and $comp_r(t,v) \Leftrightarrow \forall v_1, v_2 \in overlap_r(t,v) : v_1 \subseteq v_2 \vee v_2 \subseteq v_1$ (v_1 and v_2 form a chain).

Similarly, $overlap_w(t,v)$ is the non-empty intersection of the write view of t with v and $comp_w(t,v) \Leftrightarrow \forall v_1, v_2 \in overlap_w(t,v) : v_1 \subseteq v_2 \vee v_2 \subseteq v_1$.

We therefore need to compute the maximal views of each thread. Since only t_0 is composed of several transactions and even those don't access the same TVars, the maximal view are identical to the views generated in section 6.1.

Using the shortened notation, we have:

$M(t_0)$	$\{\{(r,\texttt{bob})\},\{(r,\texttt{jill})\}\}$
$M_r(t_0)$	$\{\{\texttt{bob}\},\{\texttt{jill}\}\}$
$M_w(t_0)$	$\emptyset$
$M(t_1)$	$\{(r,\texttt{bob}),(w,\texttt{bob}),(r,\texttt{jill}),(w,\texttt{jill})\}$
$M_r(t_1)$	$\{\texttt{bob},\texttt{jill}\}$
$M_w(t_1)$	$\{\texttt{bob},\texttt{jill}\}$
$M(t_2)$	$\{(r,\texttt{bob}),(w,\texttt{bob}),(r,\texttt{jill}),(w,\texttt{jill})\}$
$M_r(t_2)$	$\{\texttt{bob},\texttt{jill}\}$
$M_w(t_2)$	$\{\texttt{bob},\texttt{jill}\}$

To check the view safety property for the program in example 11, we must have

$$\begin{array}{ll} & [comp_w(t_2, M_r(t_0)) \wedge comp_w(t_2, M_w(t_0)) \wedge comp_r(t_2, M_w(t_0)) \\ \bigwedge & comp_w(t_1, M_r(t_0)) \wedge comp_w(t_1, M_w(t_0)) \wedge comp_r(t_1, M_w(t_0)) \\ \bigwedge & comp_w(t_1, M_r(t_2)) \wedge comp_w(t_1, M_w(t_2)) \wedge comp_r(t_1, M_w(t_2))] \end{array}$$

which is equivalent to

$$\begin{cases} comp_w(t_2, M_r(t_0)) & = true \\ comp_w(t_2, M_w(t_0)) & = true \\ comp_r(t_2, M_w(t_0)) & = true \\ comp_w(t_1, M_r(t_0)) & = true \\ comp_w(t_1, M_w(t_0)) & = true \\ comp_r(t_1, M_w(t_0)) & = true \\ comp_w(t_1, M_r(t_2)) & = true \\ comp_w(t_1, M_w(t_2)) & = true \\ comp_r(t_1, M_w(t_2)) & = true \end{cases}$$

Namely,

$$\begin{cases} (\forall v_1, v_2 \in overlap_w(t_2, M_r(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_w(t_2, M_w(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_r(t_2, M_w(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_w(t_1, M_r(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_w(t_1, M_w(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_r(t_1, M_w(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_w(t_1, M_r(t_2)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_w(t_1, M_w(t_2)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_r(t_1, M_w(t_2)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \end{cases}$$

Since there is only one view in both t_1 and t_2, the condition is true for each intersection of views of those threads, and the system becomes:

$$\begin{cases} (\forall v_1, v_2 \in overlap_w(t_2, M_r(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_w(t_2, M_w(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_r(t_2, M_w(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_w(t_1, M_r(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_w(t_1, M_w(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ (\forall v_1, v_2 \in overlap_r(t_1, M_w(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ true & = true \\ true & = true \\ true & = true \end{cases}$$

Since $M_w(t_0) = \emptyset$, any intersection with it would also be empty and each of the corresponding condition is trivially true. The previous system can be rewritten:

$$\begin{cases} (\forall v_1, v_2 \in overlap_w(t_2, M_r(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ true & = true \\ true & = true \\ (\forall v_1, v_2 \in overlap_w(t_1, M_r(t_0)) : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)) & = true \\ true & = true \\ true & = true \\ true & = true \\ true & = true \\ true & = true \end{cases}$$

$overlap_w(t_2, M_r(t_0)) = \{M_r(t_0) \cap v \mid (v \in V_w(t_2)) \wedge (v \cap M_r(t_0) \neq \emptyset\} = \{\texttt{bob}, \texttt{jill}\}$ since $M_r(t_0) = \{\{\texttt{bob}\}, \{\texttt{jill}\}\}$ and $V_w(t_2) = \{\texttt{bob}, \texttt{jill}\}$.

Similarly, $overlap_w(t_1, M_r(t_0)) = \{\texttt{bob}, \texttt{jill}\}$.

The view safety property is therefore reduced to $\forall v_1, v_2 \in \{\texttt{bob}, \texttt{jill}\} : (v_1 \subseteq v_2) \vee (v_2 \subseteq v_1)$, which is true since there is a single view in $\{\texttt{bob}, \texttt{jill}\}$.

Hence, the view safety property is verified for the program in example 11, and the view consistency sensor won't report a warning.

Part III

Related works

Chapter 7

Problematic access patterns classification

Concurrency errors-related literature contains a few attempts at a (more or less) exhaustive classification of problematic scenarios in concurrent programs. Related detection-approaches search for occurences of those scenarios instead of testing a criterion as Moth sensors do. The present chapter presents examples of such approaches and examines if the scenarios are detected by Moth sensors.

7.1 Common transactional memory anomalies on related set of variables

The authors of [12] recommend to look for two kinds of occurence: cases of non-atomic global reads (reading two related variables in separate transactions) and cases of non-atomic global writes (writing two related variable in separate transactions) and to report them if another thread in the program may conflict with them. Another thread is said to be in potential conflict if it writes, reads or writes and reads the same variables in a transaction and could therefore either observe or cause a memory inconsistency.

The authors do not claim their classification covers the whole range of possible concurrency errors. Stale-value errors, for example, are not integrated in their classification. They only profess that most of the bugs encountered in programs can be classified as either non-atomic global writes or non-atomic global reads. Their classification of major causes does not exclude the use of complementary approaches for what they deem to be less common bug classes.

The theory behind [12] is part of Moth's groundwork and it is clear that those scenarios are targeted by the view consistency sensor. The detection of related variables and accordingly the potential for both false positive and false negative due to this detection approach is similar to that of the related Moth sensor.

7.2 Common concurrency anomalies on single variables

The authors of [15] identify three common causes of concurrency error involving a single shared variable: non atomic successive reads, non atomic successive writes and non atomic successive reads and writes. Their tool, AVIO, reports an error if another thread in the program sometimes conflicts with one of those scenarios. Another thread is said to be in potential conflict if it writes, reads or writes or reads the same variable in a transaction in such a way that it could observe or cause a memory inconsistency.

The four scenario listed in [15] can be expressed in STM Haskell as follows, assuming a is a TVar accessed in two threads (Thread 1 and Thread 2).

As a reminder, $writes(\gamma,t)$ means that thread t writes TVar γ and $psv(\gamma,t)$ means that $\exists v_1 \neq v_2 \in V(t) : (r,\gamma,\circ) \in v_1 \wedge (w,\gamma,\beta) \in v_2$ where $\beta \in \{\circ,\bullet\}$. The single-variable sensor in Moth checks that for each pair of threads, t_1 and t_2, and each Tvar γ, that $(\neg writes(\gamma,t_1) \vee \neg psv(\gamma,t_2)) \wedge (\neg writes(\gamma,t_2) \vee \neg psv(\gamma,t_1))$ and reports an error if the condition is false.

	Thread 1	Thread 2
1	$atomically(readTVar\ a)$ $atomically(readTVar\ a)$	$atomically(writeTVar\ a)$
	Not detected because $\neg writes(a,Thread\ 1) \wedge \neg psv(a,Thread\ 1)$	
2	$atomically(writeTVar\ a)$ $atomically(readTVar\ a)$	$atomically(writeTVar\ a)$
	Detected by the single-variable sensor because $writes(a,Thread\ 2) \wedge psv(a,Thread\ 1)$	
3	$atomically(writeTVar\ a)$ $atomically(writeTVar\ a)$	$atomically(readTVar\ a)$
	Not detected because $\neg writes(a,Thread\ 2) \wedge \neg psv(a,Thread\ 1)$	
4	$atomically(readTVar\ a)$ $atomically(writeTVar\ a)$	$atomically(writeTVar\ a)$
	Identical to 2 as far as Moth is concerned	

Moth's single value sensor does not detect scenario 1 as problematic because the fact that successive reads yield different values in the same thread does not indicate that a stale value is being used.

Scenario 1 is not detected either because the fact that a value is read before it is updated in another thread does not indicate that a stale value is being used in the reading thread.

The authors do not claim their classification covers the whole range of possible concurrency errors. Errors involving several variables, for example are neither integrated in their classification nor reported by their tool.

7.3 Problematic interleaving scenarios in concurrent programs

The authors of [10] identify 11 problematic interleaving scenarios (later incorporated in other articles such as [16] and [17]).

The authors of [10] claim that their classification is exhaustive and that if a program displays none of those 11 scenarios, it is *atomic-set serializable*, which means that all of the functions in the program run exactly as if there was no concurrent execution with regard to the shared variable they work with. Moth detects some of those patterns as problematic but not all of them because some can't possibly cause concurrency errors in STM Haskell programs and some others do not directly cause memory inconsistency, so reporting them would cause false warnings more than help find bugs in the code.

The 11 original scenarios cover both cases of stale-value errors and high-level data races.

7.3.1 Stale-value errors

The first five scenarios of [10] target concurrency errors relating to a single shared variable and can be expressed in STM Haskell as follows, assuming a is a TVar accessed in two threads (Thread 1 and Thread 2).

As a reminder, the single-variable sensor assumes that a couple of threads is safe for a TVar γ iff one of them does not read γ ($\neg writes$) or the other cannot cause a stale-value error for γ ($\neg psv$) (because it does not read and update γ in separate transactions).

	Scenario	Thread 1	Thread 2
1	$R_u(l)W_{u'}(l)W_u(l)$	$atomically(readTVar\ a)$ $atomically(writeTVar\ a)$	$atomically(writeTVar\ a)$
	Detected by the single-variable sensor because $writes(a, Thread\ 2) \wedge psv(a, Thread\ 1)$		
2	$R_u(l)W_{u'}(l)R_u(l)$	$atomically(readTVar\ a)$ $atomically(readTVar\ a)$	$atomically(writeTVar\ a)$
	Not detected because $\neg writes(a, Thread\ 1) \wedge \neg psv(a, Thread\ 1)$		
3	$W_u(l)R_{u'}(l)W_u(l)$	$atomically(writeTVar\ a)$ $atomically(writeTVar\ a)$	$atomically(readTVar\ a)$
	Not detected because $\neg writes(a, Thread\ 1) \wedge \neg psv(a, Thread\ 1)$		
4	$W_u(l)W_{u'}(l)R_u(l)$	$atomically(writeTVar\ a)$ $atomically(readTVar\ a)$	$atomically(writeTVar\ a)$
	Detected by the single-variable sensor because $writes(a, Thread2) \wedge psv(a, Thread1)$		
5	$W_u(l)W_{u'}(l)W_u(l)$	$atomically(writeTVar\ a)$ $atomically(writeTVar\ a)$	$atomically(writeTVar\ a)$
	Not detected but the lost update problem can't happen in STM Haskell		

Cases that are detected by the Moth single-value sensor are those where a write to a TVar may rely on a previously read value or the contrary without any guarantee the value is still accurate.

Scenario 2 is not detected by the sensor because successive reads yielding different values in the same thread will not affect the value of the TVar. Unless a is also writen inside the transaction, the memory coherency will not be affected by the fact that inside the transaction the value read will depend on thread interleaving and there is no indication that the second read depends on the value returned by the first read.

Similarly, scenario 3 is not detected because there is no indication that the second write relies on the value that was previously written or that thread 2 needs to read the "final" value of a. However, such interleaving may cause concurrency errors if the programmer has erronously made such assumptions.

As a consequence, the single-variable sensor avoids some likely false positive caused by detection approaches based on the classification of [10] but may generate false negatives in cases correctly detected in [10].

The single-variable sensor also leaves out the fifth pattern because it is spurious for STM Haskell: low-level data-races are excluded by the STM interface.

The Moth single-variable sensor seems to produce less spurious warnings than a brute transposition of the detection of the problematic scenario classification of [10] in STM Haskell would. However, those scenario were not developed in a context where atomicity and isolation are already guaranteed, which complicates any comparison between the two approaches.

Finally, both the approach in [10] and Moth's single-variable sensor are underreporting because neither considers cases where a stale value is used to update another shared variable than the one its value is derived from.

7.3.2 High-level data races

The last six scenarios of [10] target concurrency errors relating to coherency within a related set of shared variables can be expressed in STM Haskell as follows assuming a and b are related TVars accessed in two threads (Thread 1 and Thread 2).

A major difference between Moth and the senario-based approaches is that these rely on a external information to know that a and b are related while Moth will have to get that information from the source code alone, which biases any comparison between the two kinds of approaches.

As a reminder, $overlap_r(t,v)$ is the non-empty intersection of the read view of a thread t with another view sv, $overlap_w(t,v)$ is the non-empty intersection of the write

view of t with v, $comp_r(t,v) \Leftrightarrow \forall v_1, v_2 \in overlap_r(t,v) : v_1 \subseteq v_2 \vee v_2 \subseteq v_1$ (v_1 and v_2 form a chain) and $comp_w(t,v) \Leftrightarrow \forall v_1, v_2 \in overlap_w(t,v) : v_1 \subseteq v_2 \vee v_2 \subseteq v_1$. Two threads are deemed compatible by the view consistency sensor if each of them is read compatible with the write maximal view of the other and write compatible with both the read and write maximal views of the other.

	Scenario	Thread 1	Thread 2
6	$W_u(l_1)W_{u'}(l)W_{u'}(L-l)W_u(l_2)$	$atomically(writeTVar\ a)$ $atomically(writeTVar\ b)$ $atomically(writeTVar\ a)$ $atomically(writeTVar\ b)$	$atomically(writeTVar\ a$ $writeTVar\ b)$ $atomically(writeTVar\ b$ $writeTVar\ a)$
	Detected by the view consistency sensor because $\neg comp_w(Thread\ 1, M_w(Thread\ 2))$		
7	$W_u(l_1)W_{u'}(l_2)W_u(l_2)W_{u'}(l_1)$	$atomically(writeTVar\ a)$ $atomically(writeTVar\ b)$	$atomically(writeTVar\ a)$ $atomically(writeTVar\ b)$
	Not detected because a and b are not detected as related variables		
8	$W_u(l_1)R_{u'}(l)R_{u'}(L-l)W_u(l_2)$	$atomically(writeTVar\ a)$ $atomically(writeTVar\ b)$ $atomically(writeTVar\ a)$ $atomically(writeTVar\ b)$	$atomically(readTVar\ a$ $readTVar\ b)$ $atomically(readTVar\ b$ $readTVar\ a)$
	Detected by the view consistency sensor because $\neg comp_w(Thread\ 1, M_r(Thread\ 2))$		
9	$R_u(l_1)W_{u'}(l)W_{u'}(L-l)R_u(l_2)$	$atomically(readTVar\ a)$ $atomically(readTVar\ b)$ $atomically(readTVar\ a)$ $atomically(readTVar\ b)$	$atomically(writeTVar\ a$ $writeTVar\ b)$ $atomically(writeTVar\ b$ $writeTVar\ a)$
	Detected by the view consistency sensor because $\neg comp_r(Thread\ 1, M_w(Thread\ 2))$		
10	$R_u(l_1)W_{u'}(l_2)R_u(l_2)W_{u'}(l_1)$	$atomically(readTVar\ a)$ $atomically(readTVar\ b)$	$atomically(writeTVar\ a)$ $atomically(writeTVar\ b)$
	Not detected because a and b are not detected as related variables		
11	$W_u(l_1)R_{u'}(l_2)W_u(l_2)R_{u'}(l_1)$	$atomically(writeTVar\ a)$ $atomically(writeTVar\ b)$	$atomically(readTVar\ b)$ $atomically(readTVar\ a)$
	Not detected because a and b are not detected as related variables		

Scenarios 7, 10 and 11 are not detected: Moth doesn't know that *a* and *b* are logically related because they are never accessed atomically in a single transaction.

On the other hand, Moth's view consistency sensor correctly detects all the scenarios where a legitimate usage can be observed.

As stated above, the fact that Moth detects the set of related variables automatically biases comparisons against it. Indeed, it is more vulnerable to underreporting if the erroneous usage of transactions is consistent than tools relying on user-provided informations. That weakness should however be weighed against the interest of a tool that doesn't provoke any extra workload.

Chapter 8

Concurrency anomaly detection

Different approaches aimed at detecting potential concurrency anomalies in the code of a program exist in the literature. They differ on the type of anomaly they target, the language of the programs they test and of course their methodology and more specifically how much user input they require.

The current chapter briefly discusses examples of such approaches and the possibility of adapting them or some of their ideas as aditionnal sensors for Moth in Haskell. Not all existing concurrency anomaly-related articles are discussed here: we focused on a few articles that are representatives of a "family" of approaches.

8.1 Concurrency anomaly detection not applicable to STM Haskell

Most of the concurrency anomaly-related literature focuses on error types that are not applicable in the context of STM Haskell because Haskell type systems and/or the STM interface make their occurence impossible.

For example, low-level data race detection ([18], [19], [20], [21]) and checking if sections defined as atomic by the programmer truly are ([19],[39],[40]) are useless for STM Haskell because the Haskell type system already guarantees that no unprotected access to STM variable is possible. Similarly, deadlock detection is useless in the context of STM.

8.2 Approaches based on user-provided annotations and invariants

The last section briefly addressed the subject of branches of the concurrency error detection that hold no interest in the contect of the transposition of Moth to STM Haskell.

Some other approaches found in the literature are more subtly incompatible with that of Moth because the major difference rests on the fundamental choice whether or not to make the detection fully automated (by not asking the user to document his design choices). Some such approaches will be discussed in the current section.

Lots of the work that has taken place in recent years in concurency anomalies detection for STM Haskell precisely aim to allow the programmer to define what it means for its data to be consistent.

For example, [22] asks the programmer to define a set of invariants and provides a framework using a theorem prover to check, for each atomic section, that if the set of invariant holds at the beginning of the atomic section, it still does at its end. It thus proves that none of the atomic sections exposes the variable in an inconsistent state, making undetected memory incoherency impossible if the set of invariant is adequately defined.

Similarly, [23] defines a framework checking what it calls "contracts" for pure calculations and [24] extends that approach to a concurrent environment. The extension is achieved by asking programmers to write boolean functions expressing what a consistent state is for their TVars and defining a procedure that transforms the STM Haskell program into a completely pure Haskell program whose contract can be checked using the framework of [23].

Other approaches using user-provided annotations exist both in Haskell and in other languages. However, while such approaches are far more precise and efficient than the purely automatic detection approach chosen by Moth authors, they require additional input from the programmer and therefore more work from him.

The authors of [25] go farther and extend STM Haskell with a check function, $\texttt{check} :: \texttt{STM a} -> \texttt{STM}()$ that takes an STM computation testing an invariant (to be defined by the programmer as a concurrency property relating several variables) and adds it to a global set of invariants for the program. The global set of invariant for the program must hold at the end of each transaction (even if for efficacity sake only those invariants using variable written inside the transaction will actually be checked) or the transaction won't be allowed to commit. That approach is similar to the ones in [22] and [24] but makes the invariant checker a part of the language instead of a separate tool.

Since the programmer defines what it means for its variables to be consistent, as long as the invariants are adequately defined, such tools or language extensions cause few false negatives and no false positive. Indeed, they will report all invariant violations but only actual invariant violations at the end of a transaction instead of violations of "general good practices criteria" as Moth does. However, since not all programmers can be bothered with the work overload that comes with providing invariants, automated tools such as Moth are useful despite being unable to compete with the superiority of the annotation approach in terms of precision.

8.3 Approaches requiring no user input

The preceding section presented detection approaches using user-provided annotations or invariants. The current section present some examples of approaches similar to Moth in the sense that they are based on the idea of a fully automated tool that detects anomalies from the source code alone. Their methodology is analysed to consider whether it could be used in Moth sensors.

An approach requiring no addition to the source code must necessarily define general safety rules in the context of concurrency, then express this criterion as a testable property. Such an approach does not, of course allow as many nuances as user-provided annotations: quite often the testable property will be too demanding in some cases and not demanding enough in others, resulting in both false positives and false negatives.

8.3.1 Run-time error detection

The tool presented in [41] aims at detecting logically atomic sections that are not properly synchronized when an particular execution causes a violation of the expected atomicity and to roll back the code parts involved in the problem to rerun them serially.

The tool identifies logically atomic sections without analysing existing synchronisation by making two assumptions: all the operations in a logically atomic section are interrelated and a variable written inside an atomic section is never read again inside that atomic section because if the programmer decides to read a variable it means that he expects its value may have been modified by a concurrent thread.

Run-time error detection approaches such as the one presented in [41] are fundamentally incompatible with that of Moth. Indeed, they only aim at controlling damage caused by errors remaining in programs when they are run by end users while Moth attempts to help the programmer find logical mistakes in its code so that he can correct them.

8.3.2 Dynamic detection

The tool presented in [15] detects potential concurency anomalies similar to those targeted by the single value sensor. Specifically, it aims at detecting successive operations on a single shared value that the programmer wrongly assumes to run without interference from other threads but hasn't synchronised properly.

The four scenarios the tool targets are discussed in section 7.2. The first basic assumption behind the methodology is that most wrongly applied synchronizations take one of these four forms. The second is that groups of poorly synchronised operations will execute atomically in most runs of the program (since concurrency errors only manifest themselves in specific thread interleaving) but not all of them. Therefore, a high number of runs of a program will enable to dectect both the expected behaviour

(as the statically probable case) and the faulty synchronization through the occurence of abnormal cases.

Dynamic approaches such as the one presented in [15] aren't compatible with the approach chosen in Moth because Moth doesn't run the programs it examines and does not rely on probabilistic occurence of thread interleaving.

Aside from that incompatible "statistical approach", we saw in section 7.2 that the tool targets either scenarios that are already detected by the single-value sensor or scenarios that are not likely to cause memory incoherency in STM Haskell programs.

8.3.3 Static error detection

The tool presented in [1] reports stale-value usage if a local value is initialized in an atomic section and used in another atomic section. It does so by adding a boolean variable $\gamma IsStale$ to each local copy γ of a shared variable. Each time an atomic section starts, all $IsStale$ variables become true and $\gamma IsStale$ only becomes false when the γ is initialized or updated inside the atomic section. The tool then reports a warning each time a variable is used and the corresponding $IsStale$ is true.

That tool targets stale-value errors that are not detected by the single -variable sensor such as usage of a stale value to update another TVar than the one the local copy originates from. Accordingly, a sensor based on that idea should be added to Moth to detect such stale value errors. That sensor should incorporate a test to check that another thread writes the TVar the local variable value originates from to filter out spurious warnings.

To add such a sensor, the view generation procedure and the existing sensors need to be modified. The notion of access should be adapted to consider local copies of TVar and the single value and view consistency sensors should be adapted to exclude such accesses when analysing a view.

The extension of the notion of access could be done simply by defining an *access* $a \in \mathcal{A}$ as $(\alpha, \gamma, \beta, \lambda)$ where

- $\alpha = r$ if a is a read access to a TVAr γ;
- $\alpha = w$ if it is a write access to a TVAr γ,
- $\alpha = c$ if it is a copy from a TVar λ into a local value γ
- $\alpha = u$ if γ is used as an argument in a write to a TVar λ (where λ needs not be the TVar the value of γ originates from).

γ is the accessed variable (TVar or not) and β helps keep a "use-define" relation for

each accessed variable as follows:

If $\alpha = r$ and the value of γ will later be overwritten inside the same transaction then $\beta = \bullet$;
If $\alpha = r$ and the value of γ will not be overwritten later inside the same transaction then $\beta = \circ$;
If $\alpha = w$ and the value of γ was read before in the same transaction then $\beta = \bullet$;
If $\alpha = w$ and the value of γ was not read before in the same transaction then $\beta = \circ$;
If $\alpha = c$ and the value of γ will later be overwritten inside the same transaction then $\beta = \bullet$;
If $\alpha = c$ and the value of γ will not be overwritten later inside the same transaction then $\beta = \circ$;
If $\alpha = u$ and the value of γ was written before in the same transaction then $\beta = \bullet$;
If $\alpha = u$ and the value of γ was not written before in the same transaction then $\beta = \circ$;

The additional sensor would then report a warning if for a pair of threads t_1 and t_2 and a TVar λ, $\exists v_1 \neq v_2 \in V(t_1) : (u, \gamma, \circ, \mu) \in v_1 \wedge (c, \gamma, \circ, \lambda) \in v_2$ and $writes(\lambda, t_2)$.

The tool presented in [11] targets local copies of shared variables made in an atomic section and used in another. It proceeds by giving id numbers to each atomic section in the program and by keeping track of the id of the section from which the value of a copy originates to compare it with the current section id when the copy is read.

Despite the fact that Moth does not use id numbers for transactions, the same kind of test could be used in a sensor to track propagation of stale-values undetected by the single-value sensor. However the sensor should incorporate a test to check that another thread writes the TVar the local variable value originates from, to avoid some of the spurious warnings the original tool produces.

However, since the idea behind the criterion is the same that the one in [1] despite the differences in the way it is expressed and tested, if a sensor based on that article is added as discussed above, there is no need to add another one based on [11].

The tools found in the literature that are compatible with the approach chosen in Moth only seem to target stale-value errors. That limitation is not, however, really problematic because the single-variable sensor was found to leave out whole classes of bugs while the limits of the view coherency sensor are mostly due to the choice of making the detection of related variables purely automated.

Conclusion

We have have analysed how the Moth tool, developed for Java programs, could be adapted to STM Haskell. Moth is an automatic detector for consistency problems emanating from internal concurrency between threads interacting in a single process.

The Moth tool is automatic and requires no user input in addition to the source code: it works by infering logical relationship between variables and operations. It infers which shared variables are logically related based on which shared variables are accessed together in some part of the code, then reports potentially problematic partial access to those variables. It also infers which operations are logically related, based on usage of the same shared variable in successive operations, and reports potentially problematic lack of atomicity between those operations.

The context of STM Haskell have both assets and drawbacks for the Moth analysis: the peculiarities of the Haskell type system make Moth self-sufficient and faster, but it increases the number of false negative.

Finally, we have proposed an extension of Moth by adding another sensor to broaden the range of relationships between operations detected by Moth, thus limiting the number of false negative regarding stale-value errors.

However, regardless of how much such a tool could be enhanced, it can never claim the soundness and completeness of a tool checking for high-level data races based on user-provided invariants.

Moth is nonetheless a very useful tool because it can find a lot of concurrency errors at limited cost for the user, and not all programmers would agree to spend time provinding formal invariants for all their `TVars`. Much in the same way that formal proof of programs has not made testing obsolete, the existence of a superior technique therefore does not detract in any way from the usefulness of the Moth tool in the context of STM Haskell.

Bibliography

[1] M. Burrows and L. K, "Finding stale-value errors in concurrent programs," Tech. Rep., May 2002, retrieved (November 2012). [Online]. Available: research.microsoft.com/en-us/um/people/leino/papers/krml107.pdf

[2] V. Pessanha, R. J. Dias, J. M. Lourenço, E. Farchi, and D. Sousa, "Practical verification of high-level dataraces in transactional memory programs," in *Proceedings of 9th the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, New York, NY, USA, July 2011, pp. 26–34, retrieved (October 2012). [Online]. Available: www.research.ibm.com/haifa/Workshops/../p26-pessanha.pdf

[3] W. Vanhoof, "2013-pfl," retrieved (February 2013). [Online]. Available: http://webcampus.fundp.ac.be/claroline/document/document.php?cmd=exChDir&file=L3N5bGxhYnVzL1BhcnRpZV9QRkw$%$3D&cidReset=true&cidReq=INFOB316

[4] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2009.

[5] C. Artho, K. Havelund, and A. Biere, "High-level data races," in *Journal on software testing, verification and reliability: Special Issue: VVEIS 2003—Workshop on Verification and Validation of Enterprise Information Systems*, vol. 13(4), December 2003, p. 207–227, retrieved (October 2012). [Online]. Available: http://staff.aist.go.jp/c.artho/papers/stvr03.pdf

[6] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '95. New York, NY, USA: ACM, 1995, pp. 204–213, retrieved (November 2012). [Online]. Available: www.cse.ohiostate.edu/~agrawal/788-su08/.../shavit95software.pdf

[7] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the 22nd annual ACM symposium on Principles of distributed computing*, 2003, retrieved (November 2012). [Online]. Available: cs.brown.edu/courses/csci1610/papers/stm.pdf

[8] T. Harris, A. Cristal, O. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero, "Transactional memory: An overview," *IEEE Micro*, vol. 27, no. 3, pp. 8–29, May 2007, retrieved (April 2013). [Online]. Available: http://dx.doi.org/10.1109/MM.2007.63

[9] A. Discolo, T. Harris, S. Marlow, S. Peyton Jones, and S. Singh, "Lock -Free Data Structures using STMs in Haskell," in *Eighth International Symposium on Functional and Logic Programming*, 2006, retrieved (February 2013). [Online]. Available: http://research.microsoft.com/~{}simonpj/papers/stm/lock-free.htm

[10] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '06, vol. 41, no. 1. New York, NY, USA: ACM, 2006, pp. 334–345, retrieved (January 2013). [Online]. Available: http://doi.acm.org/10.1145/1111037.1111067

[11] C. Artho, K. Havelund, and A. Biere, "Using block-local atomicity to detect stale value concurrency errors," in *Proc. ATVA '04*. Springer, 2004.

[12] B. Teixeira, J. Louren, E. Farchi, R. Dias, and D. Sousa, "Detection of transactional memory anomalies using static analysis," in *Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, ser. PADTAD '10. New York, NY, USA: ACM, 2010, pp. 26–36, retrieved (November 2012). [Online]. Available: http://docentes.fct.unl.pt/sites/default/files/joao-lourenco/files/p3-teixeira.pdf

[13] A. Levy, *Basic Set Theory*. Dover Publications, 2002.

[14] Retrieved (April 2014). [Online]. Available: http://en.wikipedia.org/wiki/Concurrent_Haskell

[15] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: detecting atomicity violations via access interleaving invariants," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 37–48. [Online]. Available: http://planetlab-opera-1.ucsd.edu/paper/asplos062-lu.pdf

[16] C. Hammer, J. Dolby, M. Vaziri, and F. Tip, "Dynamic detection of atomic-set-serializability violations," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 231–240.

[17] Z. Lai, S.-C. Cheung, and W. K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 235–244.

[18] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411,

1997. [Online]. Available: http://cse.iitd.ac.in/~sbansal/os/previous_years/2011/bib/savage97eraser.pdf

[19] Z. Letko, T. Vojnar, and B. Křena, "Atomrace: data race and atomicity violation detector and healer," in *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*. ACM, 2008, p. 7. [Online]. Available: http://www.cs.umd.edu/~pugh/ISSTA08/padtad2008/papers/a9-letko.pdf

[20] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar, "Healing data races on-the-fly," in *Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*. ACM, 2007, pp. 54–64. [Online]. Available: http://www.fit.vutbr.cz/~vojnar/Publications/kltuv-padtad-07.pdf

[21] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," *ACM SIGPLAN Notices*, vol. 38, no. 10, pp. 167–178, 2003. [Online]. Available: http://web5.cs.columbia.edu/~junfeng/09fa-e6998/papers/hybrid.pdf

[22] R. Demeyer and W. Vanhoof, "A framework for verifying the application-level race-freeness of concurrent programs," in *22nd Workshop on Logic-based Programming Environments (WLPE 2012)*, 2012, p. 10, retrieved (April 2013). [Online]. Available: http://www.sics.se/~matsc/ICLP2012/WLPE-proceedings.pdf#page=15

[23] D. N. Xu, S. P. Jones, and K. Claessen, "Static contract checking for haskell."

[24] R. Demeyer and W. Vanhoof, "Verification of transactions in stm haskell using contracts and program transformation," 2013, p. 47, retrieved (April 2013). [Online]. Available: http://plone.di.fc.ul.pt/places13/Members/Members/proceedings-PLACES13.pdf#page=51

[25] T. Harris and S. Peyton Jones, "Transactional memory with data invariants," in *TRANSACT '06: 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, jun 2006, retrieved (February 2013). [Online]. Available: http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/index.htm

[26] *Subtleties of Transactional Memory Atomicity Semantics*, vol. 5, no. 2, November 2006, retrieved (December 2012). [Online]. Available: acg.cis.upenn.edu/papers/cal06_atomic_semantics.pdf

[27] M. Herlihy and J. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on computer architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 289–300, retrieved (December 2012). [Online]. Available: www.cs.utexas.edu/~pingali/CS395T/.../herlihy93transactional.pdf

[28] J. Larus and C. Kozyrakis, "Transactional memory," *Commun. ACM*, vol. 51, no. 7, pp. 80–88, Jul. 2008, retrieved (March 2013). [Online]. Available: http://csl.stanford.edu/~christos/publications/2008.tm.cacm.pdf

[29] A. Tanenbaum, *Operating Systems Design and Implementation*, 3rd ed., P. Hall, Ed., Amsterdam, 2006.

[30] C. Flanagan and S. Qadeer, "Types for atomicity," in *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, ser. TLDI '03, 2003, pp. 1–12, retrieved (January 2013). [Online]. Available: www.soe.ucsc.edu/~cormac/papers/tldi03.pdf-UnitedStates

[31] C. Flanagan, K. Leino, M. Rustan, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata, "Extended static checking for java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, ser. PLDI '02. New York, NY, USA: ACM, 2002, pp. 234–245, retrieved (November 2012). [Online]. Available: www.cs.cornell.edu/courses/cs711/2005fa/papers/esc-pldi02.pdf

[32] N. Beckman, K. Bierhoff, and J. Aldrich, "Verifying correct usage of atomic blocks and typestate," in *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, ser. OOPSLA '08. ACM, 2008, pp. 227–244, retrieved (November 2012). [Online]. Available: http://doi.acm.org/10.1145/1449764.1449783

[33] C. Von Praun and Gross, "Static detection of atomicity violations in object-oriented programs," 2003, retrieved (January 2013). [Online]. Available: www.lst.inf.ethz.ch/research/publications/FTFJP.../FTFJP_2003.pdf

[34] L. Wang and S. Stoller, "Runtime analysis of atomicity for multi-threaded programs," *IEEE Transactions on Software Engineering*, vol. 32, 2006, retrieved (January 2013). [Online]. Available: ftp://ftp.cis.upenn.edu/pub/papers/lee/entcs-89-2/89.2.012.pdf

[35] M. Herlihy and J. Wing, "Linearizability: a correctness condition for concurrent objects," pp. 463–492, July 1990, retrieved (January 2013). [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/similar?doi=10.1.1.142.5315

[36] S. Peyton Jones, *Beautiful Concurrency*. O'Reilly Media, Inc., 2007, retrieved (February 2013). [Online]. Available: http://research.microsoft.com/Users/simonpj/papers/stm/index.htm

[37] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 48–60, retrieved (February 2013). [Online]. Available: http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/index.htm

[38] R. Demeyer and W. Vanhoof, "Proper granularity for atomic sections in concurrent programs," 2011, p. 244, retrieved (April 2013). [Online]. Available: http://users.dsic.upv.es/~gvidal/german/informal_proceedings_lopstr2011.pdf#page=252

[39] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *ACM SIGPLAN Notices*, vol. 38, no. 5. ACM, 2003, pp. 338–349. [Online]. Available: http://slang.soe.ucsc.edu/cormac/papers/pldi03.pdf

[40] C. Flanagan and S. N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 256–267, 2004. [Online]. Available: http://www.cis.upenn.edu/~lee/04cis700/papers/FF04.pdf

[41] M. Xu, R. Bodík, and M. D. Hill, "A serializability violation detector for shared-memory server programs," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 1–14. [Online]. Available: http://www.cs.berkeley.edu/~bodik/research/pldi05-svd.pdf